

COMP6451 T1 2022

Assignment 2

Haozhe Chen

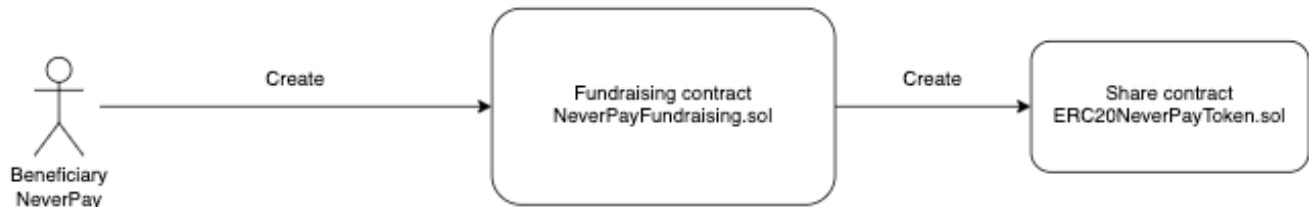
z5142012

Part 1a: Contract structure overview

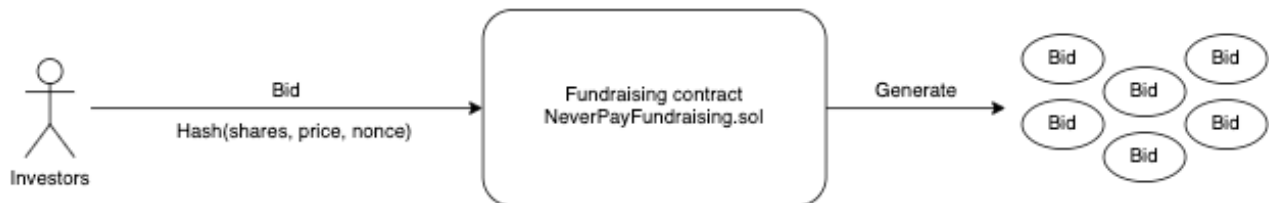
This assignment can be divided into two parts of contract:

1. Share contract (**ERC20NeverPayToken.sol**) used ERC20 standard interface
2. Fundraising contract (**NeverPayFundraising.sol**) which is used to bid and reveal.

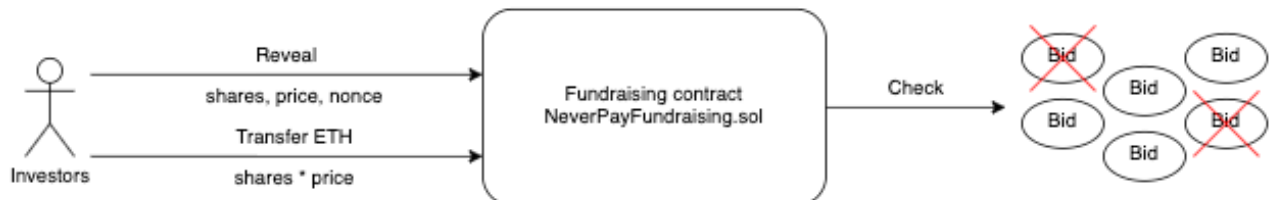
[Round 0] Initial



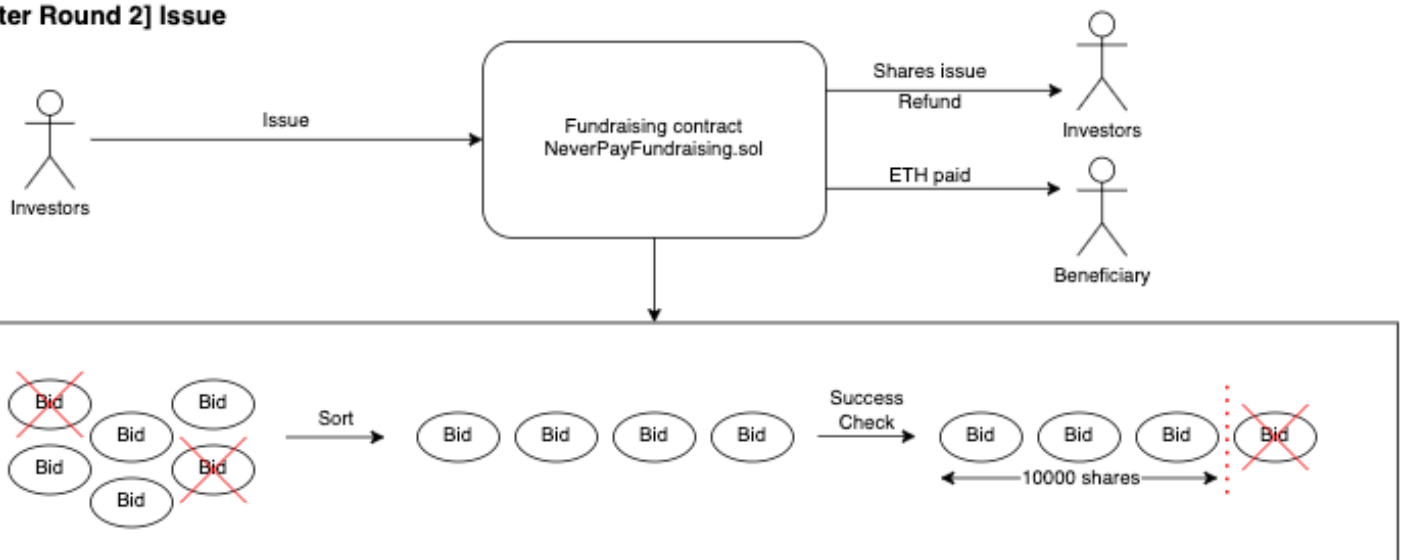
[Round 1] Bid



[Round 2] Reveal



[After Round 2] Issue



[Round 0] Initial

In this round, beneficiary (aka NeverPay company) will deploy the fundraising contract (NeverPayFundraising.sol) into the blockchain. In the fundraising contract's constructor, the share contract (ERC20NeverPayToken.sol) will be initiated. Before this round end, the fundraising contract should keep (lock) 10000 tokens.

[Round 1] Bid

In this round, investors are able to call bid function to make a blinded bid. The parameter should be a hash value = $\text{keccak256}(\text{share}, \text{value}, \text{nonce})$.

[Round 2] Reveal

In this round, investor will call reveal function to open their bid in round1 and pay to the contract. The parameters are shares, value and nonce.

If hash value of these three parameters is equal to bid record in round1 and amount paid is larger than $\text{price} * \text{shares}$, this reveal is successful. Otherwise, this reveal is failed.

[After Round 2] Issue

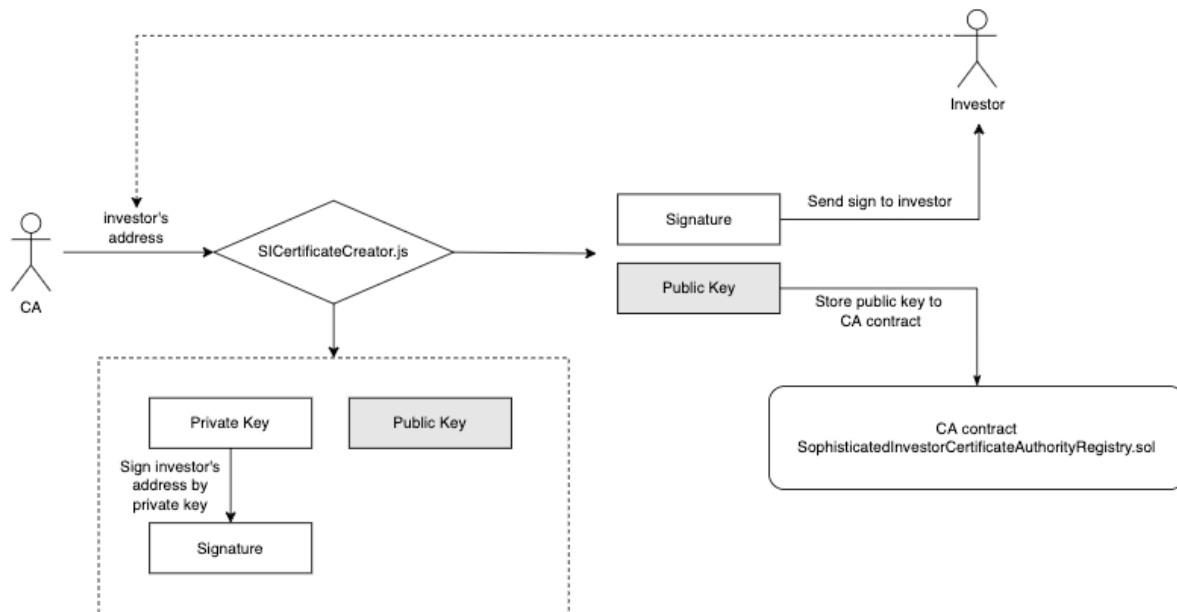
This round is used to let every investor to get their shares and refunds. Beneficiary can also call this function to get ETH collected during the last round (only successful bid). The issue function will firstly sort every bid (reveal successfully in round2), then the successful bid's owner will get the shares, failed bid's owner will get the refunds.

Part 1b (stretch): CA Contract structure overview

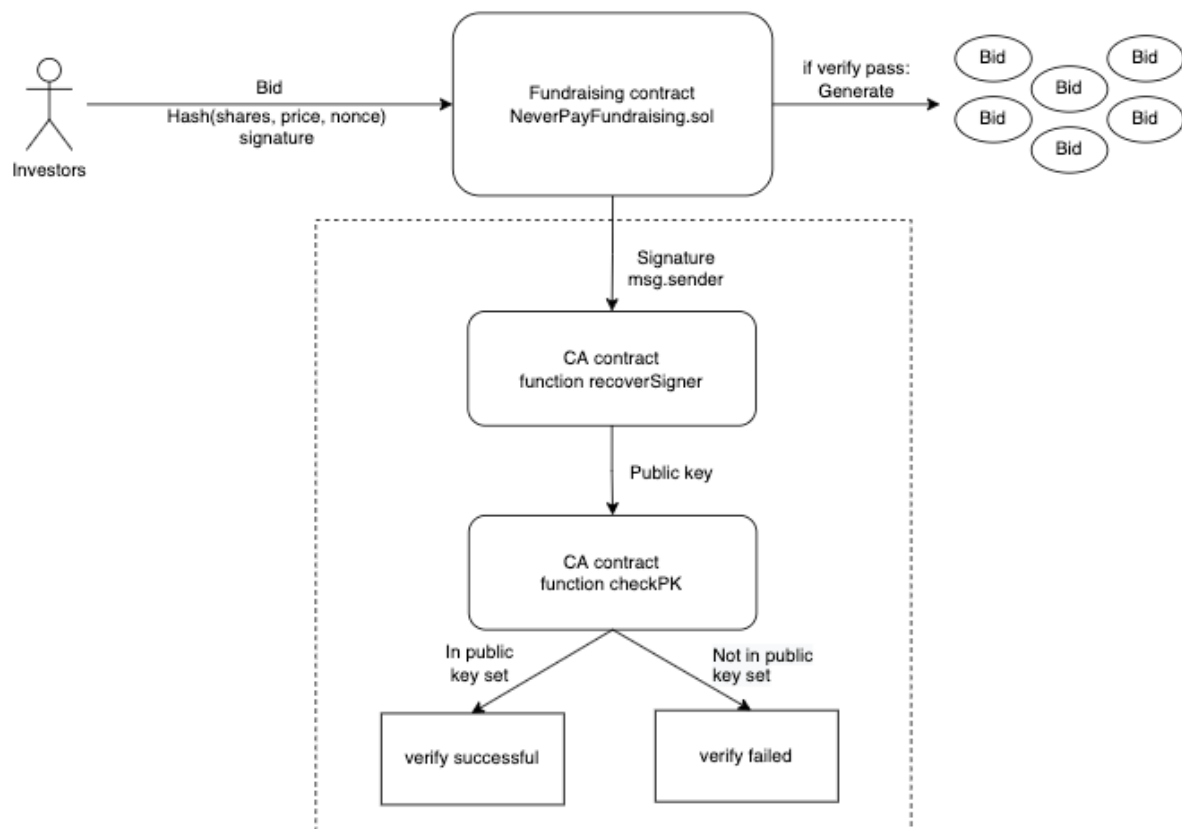
CA contract contains two main parts:

1. Off-chain: CA use node.js script to sign investor's address
2. Fundraising contract uses CA contract to verify signature.

Signature and key pair generating



Signature verification when in round1



[off-chain script] Signature generator

CA (ASIC) will use this node.js script to create a key pair, the private key will be used to sign investor' s address by script. After that, signature will be sent to investor and public key will be added into public key set of CA contract (SophisticatedInvestorCertificateAuthorityRegistry.sol).

[Round1] Signature verification

When investors bid, their provide not only hash value but also signature (get from CA) to bid function. Bid function will then pass signature and investor' s address (by msg.sender) to **recoverSigner** function of CA contract.

In recoverSigner function, it will call **ecrecover** to get public key corresponding to private key which signed signature before.

CA contract should then use **checkPK** function to check whether this public key is in public key set of CA contract.

If publicKey in public_key_set:

Verify pass

Else:

Verify failed

Part 2a: Data model (Token contract)

Token contract: ERC20NeverPayToken.sol		
Property	Type	Description
name	uint256	Token' s name
decimals	uint8	How many decimals to show
symbol	string	Token' s identifier
balances	mapping address => uint256	user_adress => balances User' s balances
totalSupply	uint	Token' s total supply

The token contract is followed by ERC20 token interface. Every users' balance is stored by a mapping structure explicitly.

Function: transfer		
Parameters	Type	Description
_to	address	Receiver' s address
_value	uint256	Transfer amount
Return type	Description	
bool	Transfer successfully or failed	

This function is used to let sender to transfer money to certain address.

Function: transferFrom		
Parameters	Type	Description
_from	address	Sender' s address
_to	address	Receiver' s address
_value	uint256	Transfer amount
Return type	Description	
bool	Transfer successfully or failed	

This function is also used to transfer money. However, this function should be called by receiver to initiate a transaction from a certain address. Which also need sender' s to approve this transaction with approve function.

Function: approve		
Parameters	Type	Description
_spender	address	Receiver' s address
_value	uint256	Approve transfer amount
Return type	Description	
bool	Approve successfully or failed	

This function should be called by sender to approve amount of money transferred by certain address. In case of receiver 'stole' money from sender (can initiate transfer with any amount from any address).

Function: balanceOf		
Parameters	Type	Description
_owner	address	Owner' s address
Return type	Description	
uint256	Owner' s balance	

This function is used to get the balance of certain address.

Function: allowance		
Parameters	Type	Description
_owner	address	Sender' s address
_spender	address	Receiver' s address
Return type	Description	
uint256	Value approved by sender to transfer.	

This function is used to check the amount of value approved by sender to transferred to receiver.

Part 2b: Data model (Fundraising contract)

Token contract: NeverPayFundraising.sol		
Property	Type	Description
token	ERC20NeverPayToken	Token' s contract address.
beneficiary	address payable	Beneficiary' s address.
bidEnd	uint	timestamp of round1 end: 22/04/2022
revealEnd	uint	timestamp of round2 end: 27/04/2022
bids	mapping(address => mapping(bytes32 => bool))	addr => (bid_hash_value => valid) Whether this bid is able to reveal.
refunds	mapping(address => uint)	addr => refund_amount Record refund amount of every investor.
validBid	struct: address addr uint shares uint price uint bid_order	Structure of a valid bid. Contain bidder' s address, bid' s share and price, and order of this bid.
validBids	validBid[]	List contain every valid bid.
bidOrder	mapping(bytes32 => uint)	bid_hash_value => order Mapping to keep order of each bid.
order	uint32	Record global order of bid.
issued	mapping(address => bool)	addr => issued Mapping to record whether this investor has called issue function before.

This contract is used to fundraise the ETH from investors. Several mapping structures are used to keep bid' s information:

[bids] is used to record whether this bid is able to reveal, it locates different bids by their investors' address and hash_value. The value is true when bid is been created. When investors withdraw this bid or reveal this bid, it will be marked as false.

[refunds] is used to record refund value of every investor at round1 & 2.
E.g., bid withdraw, reveal failed (mismatched bid info, insufficient payment)

[bidOrder] is used to keep every bid' s order at round1, so that when investors reveal their bid, contract can get bid' s order at round1 by bid' s hash_value.

[issued] is used to check whether certain investor has been issued before, to prevent the investors issuing multiple times to get extra shares and refund.

After investors revealing their bid, contract will use **validBid** structure to store the bids, it contains every detail information of this bid to help contract initiate shares transfer of ETH refund after round2. The **order** property of **validBid** structure will be used to keep bids' order when sort if there are multiple bids with same price.

Function: bid		
Parameters	Type	Description
_bindedBid	bytes32	h(shares, price, nonce)

Investors will use this function to make a bid in round1. Since this fundraising contract is a blinded bid contract. User should use a keccak256 hash function to encrypt their bids information, so that other investors cannot get bids' detail. And in case of bids with same shares and price, hash function also contains a different nonce given by different investors.

Function: withdrawBid		
Parameters	Type	Description
_bindedBid	bytes32	h(shares, price, nonce)

Investors will use function to withdraw a bid. It will modify **bids[msg.sender][_bindedBid]** to false, so that this bid cannot be revealed in round2.

Function: reveal		
Parameters	Type	Description
_shares	uint	Shares contained in bid in round1.
_price	uint	Price contained in bid in round1.
_nonce	bytes32	Nonce used to calculate hash with price and shares.

During the round2, investors will give the shares, price, and nonce to reveal function. If hash value calculated by three parameters can be founded in **bids** mapping (from round1), this reveal will be counted as a valid reveal and recorded by **validBids** list.

Function: issue		
Parameters	Type	Description

This function should be called by investors or beneficiary after round2. It will firstly sort the validBids list, so that some potentially successful bids will become invalid bids because total shares may greater than 10000. At the end of this function, investor can get their shares or refunds depends on their bids' status. Beneficiary can call this function to calculate how much ETH collected by fundraising and get ETH transfer from the contract.

Part 2c: Data model (CA contract)

Token contract: SophisticatedInvestorCertificateAuthorityRegistry.sol		
Property	Type	Description
publicKeys	Mapping(address => bool)	Mapping to store public key.
ASIC	address	ASIC' s address

This contract should be initiate by ASIC, it can use this contract to manage the public key list. Fundraising contract can use **recoverSigner** function inside CA contract to verify public key of signature.

Function: addPK		
Parameters	Type	Description
pk	address	Public key

This function is strictly called only by ASIC, it can use this function to add public key into public key set.

Function: removePK		
Parameters	Type	Description
pk	address	Public key

This function is strictly called only by ASIC, it can use this function to remove public key from public key set.

Function: checkPK		
Parameters	Type	Description
pk	address	Public key

This function can be used to check whether certain public key is contained in public key list. Fundraising contract can use it during verification.

Function: recoverSigner		
Parameters	Type	Description
sig	btyes	Signature
addr	address	Investor' s address

This function is used to recover public key of signature. Function will use **addr** to restore hash value of investor' s address, so that **ecrecover** can use **hash(investor.address)** to recover investor' s address.

Part 3a: Off-chain computation (bid encrypt)

As discussed in previous part, in round1, investors need to submit a hash value calculated by **keccak256(shares, value, nonce)** to make a blinded bid.

```
Ass2 > truffle workstation > off_chain_scripts > JS bidEncryptScript.js > ...
1 // npm install web3
2 var Web3 = require('web3');
3 var web3 = new Web3(Web3.givenProvider)
4
5 const readline = require("readline");
6 const rl = readline.createInterface({
7   input: process.stdin,
8   output: process.stdout
9 });
10
11 console.log("----- Neverpay fundraising contract bid encrypter -----")
12 rl.question("shares: ", function(shares) {
13   rl.question("price: ", function(price) {
14     rl.question("nonce: ", function(nonce) {
15       if (price < 1) {
16         console.log("Warning: You cannot bid less than 1 Ether.");
17         process.exit(0);
18       }
19       encoded = web3.eth.abi.encodeParameters(['uint', 'uint', 'bytes32'], [shares, price, web3.utils.padLeft(web3.utils.fromAscii(nonce), 64)]);
20       hashvalue = web3.utils.soliditySha3(encoded);
21       console.log("Encrypted hash value:");
22       console.log(hashvalue);
23       bytes32Nonce = web3.utils.padLeft(web3.utils.fromAscii(nonce), 64)
24       console.log("nonce bytes32: " + bytes32Nonce);
25       process.exit(0);
26     });
27   });
28 });
```

This node.js script will help investors to get the hash value by inputting these three parameters. Investors can also get bytes32 8type of nonce because reveal function will only accept bytes32 nonce as parameters.

The script **scripts/bidEncryptScript.js** is already contained in submission file.

*Notice : Investors should run **npm install web3** to install web3 package before running the script.*

```
→ truffle_workstation git:(master) x node off_chain_scripts/bidEncryptScript.js
----- Neverpay fundraising contract bid encrypter -----
shares: 2000
price: 2
nonce: acc1
Encrypted hash value:
0xda4da95d5b263c087d9b968f63abb92723e754fbae3bc5621a4c14f9ff605e1d
nonce bytes32:
0x00000000000000000000000000000000000000000000000000000000000000061636331
→ truffle_workstation git:(master) x
```

Part 3b: Off-chain computation (Signature)

This script is used to create key pair and sign the investor's address.

```
JS SICertificateCreator.js M x SophisticatedInvestorCertificateAuthorityRegistry.sol 1
Ass2 > truffle_workstation > off_chain_scripts > JS SICertificateCreator.js > ...
1  var Web3 = require('web3');
2  var web3 = new Web3(Web3.givenProvider)
3
4  const account = web3.eth.accounts.create()
5
6  const readline = require("readline");
7  const rl = readline.createInterface({
8    input: process.stdin,
9    output: process.stdout
10 });
11 console.log("----- Sophisticated Investor Certificate Generator -----")
12 rl.question("Investor's Address: ", function(addr) {
13   var hash = web3.utils.soliditySha3(addr);
14   const encrypted = web3.eth.accounts.sign(hash, account.privateKey);
15
16   console.log("-----");
17   console.log("[Public key]: " + account.address);
18   console.log("[Private key]: " + account.privateKey);
19   console.log("-----");
20   console.log("[Signed address]: " + addr);
21   console.log("[Hashed text]: " + hash);
22   console.log("[Signature]: " + encrypted.signature);
23   console.log("-----");
24   console.log("Please send [Signature] and [Public key] to investor.")
25   process.exit(0);
26 })
```

This script will create a new Ethereum account every time been called. New account's address will be used as public key and its private key will be used as private key. It uses private key to sign investor's address which needs to be manually typed in terminal.

Why sign investor's address?

I design this strategy because contract can easily get investor's address when investor makes a bid (by msg.sender). So that when contract does verification, it can use msg.sender to re-calculate the hash value. This can prevent attacker from fabricating other's address to pass the verification because there is no way to call function by other's address (unless steal total account including private key).

Part 4: Requirements discussion

Requirement 1: Total shares = 10000

Token contract (ERC20NeverPayToken.sol) will be initiated with totalSupply = 10000 in fundraising contract's (NeverPayFundraising.sol) constructor. So that this 10000 tokens will be owned by fundraising contract before round1.

```
// Initial DDL of round1 and round2,
// initial addr of beneficiary.
constructor(address payable _beneficiary, SophisticatedInvestorCertificateAuthorityRegistry _ca) {
    bidEnd = 1650412800;
    revealEnd = 1651017600;
    beneficiary = _beneficiary;
    token = new ERC20NeverPayToken(10000, "NeverPay Tokens", 0, "NPT");
    CA = _ca;
    order = 0;
    issued[_beneficiary] = true;
}
```

Requirement 2: Minimum price per share is 1 Ether

Since this contract is design to settle account with Ether, so when investors use node.js script to encrypt their bids, decimal less than 1 will not be acceptable. Also, reveal function make a condition to double check the price of bid .

Notice: actually, uint type in solidity will not even accept decimal number, so no need to worry about price < 1.

```
→ truffle_workstation git:(master) x node off_chain_scripts/bidEncryptScript.js
----- Neverpay fundraising contract bid encrypter -----
shares: 1000
price: 0.5
nonce: hello
Warning: You cannot bid less than 1 Ether.
```

```
*/
function reveal(uint _shares, uint _price, bytes32 _nonce)
    payable
    public
    onlyAfter(bidEnd)
    onlyBefore([revealEnd])
{
    // Calculating the hash of shares, price and nonce in round2
    bytes32 hashedReveal = keccak256(abi.encodePacked(_shares, _price, _nonce));

    // Check whether these three parameters are matched with certain bid in round1
    // or price is smaller than 1 Ether.
    if (bids[msg.sender][hashedReveal] != true || _price < 1 || _price == 0) {
        // Bid not found || already been revealed
        // Refund the ETH
        refunds[msg.sender] += weiToETH(msg.value);
    } else {
```

Requirement 3: Investors anonymity

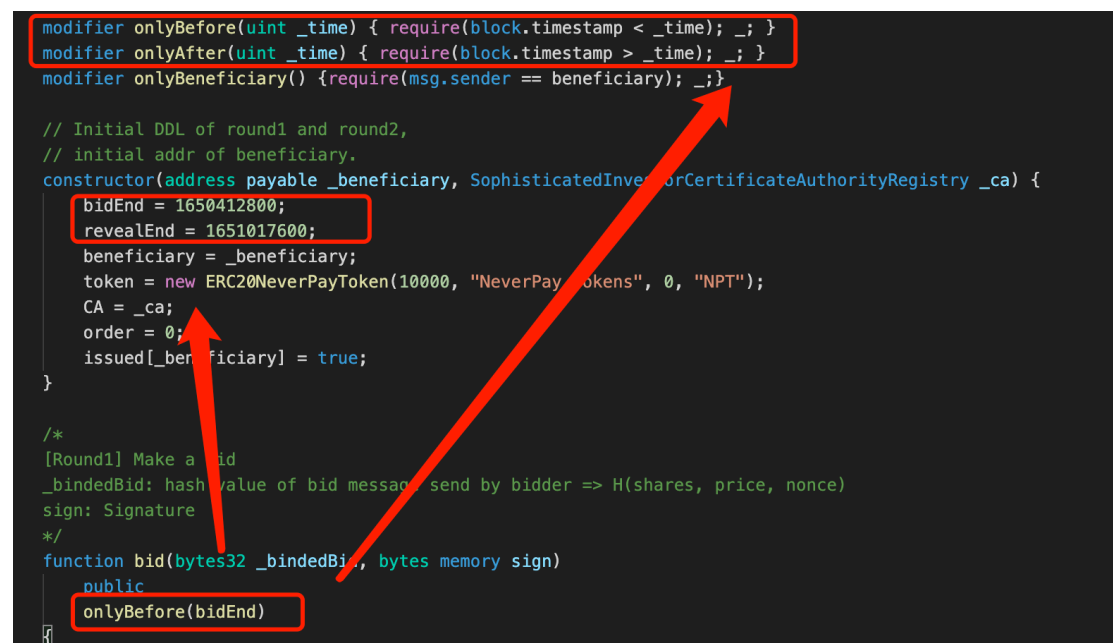
Contract will only store investors' address but not other information. Contract function can identify different investors by their Ethereum address (through **msg.sender**).

Requirement 4: Blind auction of two rounds

Fundraising contract has two main functions which are **bid** and **reveal**. **bid** is helping investors to make blinded bid and **reveal** is let user to open (reveal) their bids in round1.

Requirement 5: Deadlines setting

Deadline of each round will be hard code in contract constructor. Contract used modifier to check whether current time before every function runs



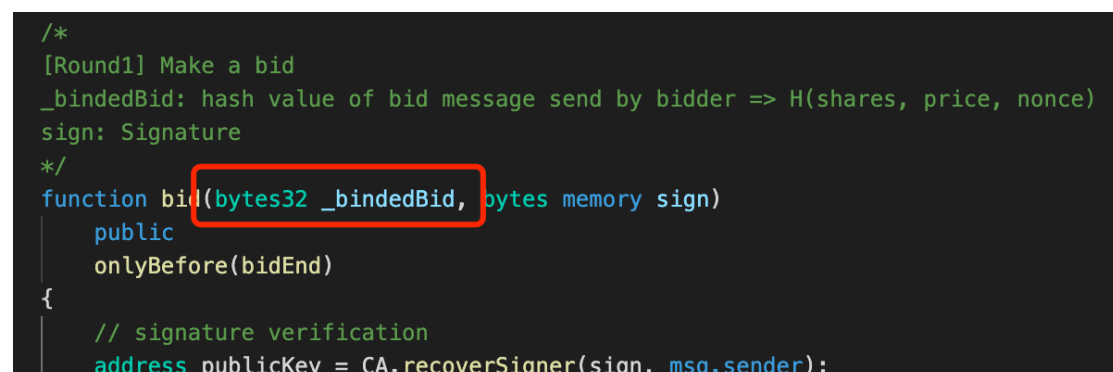
```
modifier onlyBefore(uint _time) { require(block.timestamp < _time); _; }
modifier onlyAfter(uint _time) { require(block.timestamp > _time); _; }
modifier onlyBeneficiary() {require(msg.sender == beneficiary); _;}

// Initial DDL of round1 and round2,
// initial addr of beneficiary.
constructor(address payable _beneficiary, SophisticatedInvestorCertificateAuthorityRegistry _ca) {
    bidEnd = 1650412800;
    revealEnd = 1651017600;
    beneficiary = _beneficiary;
    token = new ERC20NeverPayToken(10000, "NeverPay Tokens", 0, "NPT");
    CA = _ca;
    order = 0;
    issued[_beneficiary] = true;
}

/*
[Round1] Make a bid
_bindedBid: hash value of bid message send by bidder => H(shares, price, nonce)
sign: Signature
*/
function bid(bytes32 _bindedBid, bytes memory sign)
    public
    onlyBefore(bidEnd)
```

Requirement 6: Round1 bid information invisible

Investors will make bid in first round by submit a hash value calculated by shares, price and nonce to **bid** function. Which means nobody else even contract itself shall get no information about each bid. Investors can use node.js script mentioned in Part3 to get their bids' hash value.



```
/*
[Round1] Make a bid
_bindedBid: hash value of bid message send by bidder => H(shares, price, nonce)
sign: Signature
*/
function bid(bytes32 _bindedBid, bytes memory sign)
    public
    onlyBefore(bidEnd)
{
    // signature verification
    address publicKey = CA.recoverSigner(sign, msg.sender);
```

Requirement 7: Multiple bid and bid withdraw

There is no limitation of bid times in first round. Investors shall make several bids, contract can track bids by bids' hash value and investors' address.

```
function bid(bytes32 _bindedBid, bytes memory sign)
    public
    onlyBefore(bidEnd)
{
    // signature verification
    address publicKey = CA.recoverSigner(sign, msg.sender);
    require(CA.checkPK(publicKey));

    bids[msg.sender][_bindedBid] = true;
    bidOrder[_bindedBid] = order;
    order += 1;
    issued[msg.sender] = true;
}
```

Investors can also withdraw their bids by submit the hash value to **withdrawBid** function. Once bid has been withdrawn, it cannot be revealed in the next round.

```
/*
[Round1] Withdraw a bid
_bindedBid: hash value of bid message send by bidder => H(shares, price, nonce)
*/
function withdrawBid(bytes32 _blindedBid)
    public
    onlyBefore(bidEnd)
{
    // Make the corresponding bid unavailable to reveal.
    bids[msg.sender][_blindedBid] = false;
}
```

Requirement 8: Round2 reveal

In this round, investors should reveal their bids by submit the shares, prices, and nonce to **reveal** function. If the hash value calculated by these three parameters cannot be found in **bids** mapping (used to record bid in round1), this reveal will be treated as failed.

```
function reveal(uint _shares, uint _price, bytes32 _nonce)
    payable
    public
    onlyAfter(bidEnd)
    onlyBefore(revealEnd)
{
    // Calculating the hash of shares, price and nonce in round2
    bytes32 hashedReveal = keccak256(abi.encodePacked(_shares, _price, _nonce));

    // Check whether these three parameters are matched with certain bid in round1
    // or price is smaller than 1 Ether.
    if (bids[msg.sender][hashedReveal] != true || _price < 1000000000000000000) {
        // Bid not found || already been revealed
        // Refund the ETH
    }
}
```

Requirement 9: Round2 payment

Investors should pay exactly full price (shares * price) of their bids. If the payments are greater than full price or less than full price they should pay, contract will firstly record the refund amount, then investors can call **issue** function to get their refunds back after round2 finished.

```
/*
[Round2] Reveal bid
_shares: shares contain in bid in round1
_price: price contain in bid in round1
_nonce: nonce used to calculate hash with price and shares of bid in round1
*/
function reveal(uint _shares, uint _price, bytes32 _nonce)
    payable
    public
    onlyAfter(bidEnd)
    onlyBefore(revealEnd)
{
    // Calculating the hash of shares, price and nonce in round2
    bytes32 hashedReveal = keccak256(abi.encodePacked(_shares, _price, _nonce));

    // Check whether these three parameters are matched with certain bid in round1
    // or price is smaller than 1 Ether.
    if (bids[msg.sender][hashedReveal] != true || _price < 1 || _price == 0) {
        // Bid not found || already been revealed
        // Refund the ETH
        refunds[msg.sender] += weiToETH(msg.value);
    } else {
        // calculate the full price of this bid.
        uint totalPrice = _shares * _price;
        // Check whether the payment is enough for this bid.
        if (msg.value >= ETHtoWei(totalPrice)) {
            // Enough
            // Mark this bid as valid
            validBids.push(validBid({
                addr: msg.sender,
                shares: _shares,
                price: _price,
                bid_order: bidOrder[hashedReveal]
            }));
            // Refund the extra ETH
            refunds[msg.sender] += (weiToETH(msg.value) - totalPrice);
            // Make this bid unavailable to reveal
            bids[msg.sender][hashedReveal] = false;
        } else {
            // Not enough
            // Refund all ETH
            refunds[msg.sender] += weiToETH(msg.value);
        }
    }
}
```


Requirement 10: After Round2

After the round2, investors and beneficiary should call the **issue** function to check whether their bids are successful or fail. With successful bids, function will transfer shares (tokens) to investors. With failed bids, investors will get their refunds.

Function will firstly sort every valid bid (verify when reveal at round2).

```
function issue()
    public
    onlyAfter(revealEnd)
{
    // Only account which has bid before can call issue
    require(issued[msg.sender]);
    // Mark this account has already issued, in case of double calling
    issued[msg.sender] = false;

    // Sort every valid bid
    insertionSort();
}
```

Then if the function is called by beneficiary, it will calculate the Ethers collected by successful bids within range of first 10000 shares. After that, contract will transfer the Ethers amount to beneficiary's address

```
// If called by beneficiary
if (msg.sender == beneficiary) {
    uint totalETH = 0;
    // Get all ETH collected in Fundraising (Only successful bid)
    for (uint i = 0; i < validBids.length; i++){
        if (validBids[i].shares + totalShare > 10000) {
            overflowShare = validBids[i].shares + totalShare - 10000;
            totalETH += (validBids[i].shares - overflowShare) * validBids[i].price;
            break;
        } else {
            totalShare += validBids[i].shares;
            totalETH += validBids[i].shares * validBids[i].price;
        }
    }

    // Get paid
    beneficiary.transfer(ETHtoWei(totalETH));
}
```

Beneficiary part

If function is called by investors, it will find caller's bids from sorted valid bids list. If the bids are failed or partial successful, function will transfer refunds Ether back to investors. With successful bids, function will transfer corresponding shares (token) by calling **ERC20NeverPayToken.transfer** function.

```

} else {
    // If called by investor
    for(uint i = 0; i < validBids.length; i++) {
        if (validBids[i].addr == msg.sender) {
            // Find his/her bid, calculate the share and refund
            if (totalShare >= 10000) {
                // share overflow, get full refund
                refund += validBids[i].shares * validBids[i].price;
            } else {
                totalShare += validBids[i].shares;
                if (totalShare > 10000) {
                    // share partial overflow, get partial refund
                    overflowShare = totalShare - 10000;
                    share += validBids[i].shares - overflowShare;
                    refund += overflowShare * validBids[i].price;
                    totalShare = 10000;
                } else {
                    share += validBids[i].shares;
                }
            }
        }
    }
    // calculate the current share amount
    totalShare += validBids[i].shares;
    if (totalShare >= 10000) totalShare = 10000;
}

// get token transfer
if (share > 0) token.transfer(msg.sender, share);

// get refund transfer
uint refundAmount = refund + refunds[msg.sender];
if (refundAmount > 0) payable(msg.sender).transfer(ETHtoWei(refundAmount));
}

```

investors part

overflow handle

shares & refunds collect

Requirement 11: Bid sort with same price

Contract will use **Insertion sort** to sort the valid bids in **issue** function. By fetching the bid order from **bid_order** property of **validBid** structure, insertion sort can maintain the order of bids if they have same price.

```

function insertionSort()
public
{
    int len = int(validBids.length);
    int preIndex;
    validBid memory current;
    for(int i = 1; i < len; i++) {
        preIndex = i - 1;
        current = validBids[uint(i)];
        while(preIndex >= 0 && validBids[uint(preIndex)].price < current.price) {
            validBid memory temp = validBids[uint(preIndex)];
            validBids[uint(preIndex + 1)] = temp;
            preIndex--;
        }
        // Same price bids handle
        while(preIndex >= 0 && validBids[uint(preIndex)].price == current.price && validBids[uint(preIndex)].bid_order > current.bid_order) {
            validBids[uint(preIndex + 1)] = validBids[uint(preIndex)];
            preIndex--;
        }
        validBids[uint(preIndex + 1)] = current;
    }
}

```

Requirement 12: Payment refund

In round2, if bids revealed failed, refund will be temporarily record by a mapping (address -> refund amount).

When investors call **issue** function, if his/her partial successful bid becomes fail due to overflow shares, investors will get $\text{refund_amount} = \text{refund_round1} + \text{refund_round2}$ paid back to their address.

```
// If called by beneficiary
if (msg.sender == beneficiary) {
    uint totalETH = 0;
    // Get all ETH collected in Fundraising (Only successful bid)
    for (uint i = 0; i < validBids.length; i++){ ...
    }
    // Get paid
    beneficiary.transfer(ETHtoWei(totalETH));
} else {
    // If called by investor
    for(uint i = 0; i < validBids.length; i++) { ...
    }

    // get token transfer
    if (share > 0) token.transfer(msg.sender, share);

    // get refund transfer
    uint refundAmount = refund + refunds[msg.sender];
    if (refundAmount > 0) payable(msg.sender).transfer(ETHtoWei(refundAmount));
}
```

Requirement 13: ERC-20 standard

Since the shares (token) contract is strictly following the ERC20 interface, investors owned the shares can call function of **ERC20NeverPayToken.sol** to do the shares trade-off.

Requirement 14: Cost effective for NeverPay

As discussed above, the three main functions: **bid**, **reveal**, **issue** are all called by bidders or investors. And shares transfer function and refund pay back function are both contained in **issue** function. So that, every investor will pay for his/her own transaction, NeverPay aka beneficiary will not be charged by transferring Ether or shares to investors.

Requirement 15: Treat the investors uniformly

With this contract structure, investors only need to pay gas fee for their personal bids' s transaction cost and calculation because each main function is on the foundation of caller' s bids.

However, in issue function, every caller need to pay for sorting the valid bids. This issue may be resolved by using priority queue if there is more time space for further development.

Part 5: Gas analysis

In this part, we will analysis gas cost of each function in the view of ASIC, NeverPay and investors.

The gas cost records get from transaction records of Remix IDE. Gas price is 30 Gwei/gas, referenced from <https://crypto.com/defi/dashboard/gas-fees>. Gas fee is calculated by gas cost multiple gas price.

Below is the gas record of each function. It records gas cost of each function called by each character, simulates a fundraising process with only one bid (shares=20, price=2).

```
Ass2 > truffle_workstation > ≡ gas_anaysis.txt
```

	[function]	[gas cost]		[gas price]	[gas fee]
1					
2					
3	ASIC:				
4	deployed:	518575 gas		30 Gwei	0.015557259 Ether
5	add public key:	46314 gas		30 Gwei	0.00138942 Ether
6					
7	NeverPay:				
8	deployed:	3024447 gas		30 Gwei	0.09073341 Ether
9	issue:	40143 gas		30 Gwei	0.00120429 Ether
10					
11	Investor:				
12	bid:	112144 gas		30 Gwei	0.00336432 Ether
13	reveal:	120323 gas		30 Gwei	0.00360969 Ether
14	issue:	71066 gas		30 Gwei	0.00213198 Ether

CA contract (view of ASIC)

ASIC needs to pay 50w+ gas when deployed the CA contract. And every time adding public key, it also needs to pay 4.6w gas.

ASIC will no need to pay gas for signature verification, because this function should be called by other contract who need to verify the signature.

Fundraising contract (view of NeverPay)

NeverPay needs to pay 302w+ gas when deployed the fundraising contract including the initiation of ERC20 tokens contract with 10000 total supplies.

When NeverPay want to get Ether collected during the fundraising, it will call issue function to initiate a transaction from contract to NeverPay. However, this testcase only simulate one bid, the issue function will spend more gas with increasing number of bids (due to increasing computation of sort algorithm) because every time issue function is called, it sorts all current valid bids. In this testcase, NeverPay need to pay 4w+ gas to get the Ether paid.

Fundraising contract (view of investors)

Investors need to pay gas for three main functions during the fundraising. Bid will spend 11w+ gas, revel will spend 12w+ gas, issue will spend 7w+ gas. However, as mentioned above. Issue function will spend more gas with increasing number of bids.

Part 6: Ethereum platform suitability

This part will discuss suitability of the Ethereum platform for this blind fundraising contract.

Advantage:

1. Decentralized. Smart contracts have the characteristics of decentralization. In this case, both parties to the contract can conduct transactions through blockchain without the need for a third-party authority to provide a platform. This characteristic also saves the cost of both parties (gas fees are less than intermediary fees).
2. The open and traceable nature of the blockchain can ensure the transparency of transaction. In this contract, the transfer and refund of equity are automatically controlled by the contract to protect the interests of both parties.

Disadvantage:

1. The popularity of blockchain. Although the blockchain has developed to a certain extent. There are still only a very small number of companies are involved in blockchain transactions. The industry still needs some time to get involved or apply blockchain to real word business. In this assignment, if most investors are not ready to accept this kind of fundraising process, NeverPay may fundamentally lose a large number of investors.
2. Smart contract vulnerabilities. Smart contracts are still in the early stages of development, and there are many security issues that need to be regulated and pondered. The DAO attack is a typical case of contract bug. It is unknown to us whether smart contracts may have other serious loopholes in the future which may cause important property damage to both parties.
3. Regional compliance. Because cryptocurrencies are not legal tenders recognized by the government, some countries and regions may restrict cryptocurrency transactions in terms of policies, such as China. On the other hand, companies that use this method of financing may lose a large number of investors who have been forced to give up due to policy reasons before the fundraising has even begun.

Part 7: Avoid Reentrancy attacks

With the example of The DAO attack, the most important thing to watch out for is the reentrancy attack.

This fundraising contract, the only function that deals with refunds and ETH transaction is the issue function. Contract used a mapping structure to record whether this investor has already called issue function before. From the DAO attack example, issue function will firstly change the mapping record from true to false before the payment has been issued.

With this protection, attackers won't be able to call issue function recursively in their payment received function. Beneficiary can also only call issue function once.

```
/*
[after Round2] Issue bid
If investor has successful bid => get share
If investor has failed bid => get refund
Beneficiary => get ETH paid
*/
function issue()
    public
    onlyAfter(revealEnd)
{
    // Only account which has bid before can call issue
    require(issued[msg.sender]);
    // Mark this account has already issued, in case of double calling
    issued[msg.sender] = false;

    // Sort every valid bid
    insertionSort();

    uint totalShare = 0;
    uint overflowShare = 0;
    uint refund = 0;
    uint share = 0;

    // If called by beneficiary
    if (msg.sender == beneficiary) { ...
    } else {
        // If called by investor
        for(uint i = 0; i < validBids.length; i++) { ...
        }

        // get token transfer
        if (share > 0) token.transfer(msg.sender, share);

        // get refund transfer
        uint refundAmount = refund + refunds[msg.sender];
        if (refundAmount > 0) payable(msg.sender).transfer(ETHtoWei(refundAmount));
    }
}
```

issued verify

record update

refund and share transfer

Part 8: Security considerations

This part will tell investors any security considerations which they should concern.

Before round1:

Please do not leak your sophisticate investor certificate signature in any way, or others may use it to pass the sophisticate investor verification of this contract.

After round1:

Please do not leak your bid' s nonce hash value. Otherwise, others may use it to reveal in round2 which will make your bid in round1 become invalid.

After round2, before issue:

You can only call issue function once to collect your shares and refunds.

Overall warning:

Please do not leak your Ethereum' s private key. Otherwise, you may lose all your refunds or shares from this fundraising.

You can transfer your ETH and shares to a new Ethereum account to increasing your property security.

Part 9a: Truffle test instruction

Step 1: Create new folder as test workstation

```
$ mkdir workstation
```

```
$ cd workstation
```

Step 2: Install Ganache and Truffle

Install Ganache and Truffle with npm by running:

```
$ npm install Ganache
```

```
$ npm install Truffle
```

Step 3: Initial truffle environment

```
$ truffle init
```

Then copy contract and test from submission into *workstation/contracts* and *workstation/test*

Step 4: Set truffle config

Find truffle-config.js configuration file in folder. Uncomment the development part and set host and port.

```
networks: {
  // Useful for testing. The `development` name is special - truffle uses it by default
  // if it's defined here and no other network is specified at the command line.
  // You should run a client (like ganache-cli, geth or parity) in a separate terminal
  // tab if you use this network and you must also set the `host`, `port` and `network_id`
  // options below to some value.
  //
  development: {
    host: "127.0.0.1",      // Localhost (default: none)
    port: 8545,            // Standard Ethereum port (default: none)
    network_id: "*",      // Any network (default: none)
  }
  // Another network with more advanced options...
  // advanced: {
  //   port: 8777,          // Custom port
  //   network_id: 1342,    // Custom network
  //   gas: 8500000,        // Gas sent with each transaction (default: ~6700000)
  //   gasPrice: 20000000000, // 20 gwei (in wei) (default: 100 gwei)
  //   from: <address>,     // Account to send txs from (default: accounts[0])
  // }
```

Step 5: Run ganache test blockchain

Run the ganache with specific balance = 10000 ETH and specific data = 2022/04/07

```
$ ganache-cli -e 100000 -t 04/07/2022
```

Step 6: Run the test

1. Run the test of token contract:
\$ truffle test test/TestNPT.js
2. Run the test of certification contract:
\$ truffle test test/TestSIC.js
3. Run the test of fundraising contract:
\$ truffle test test/TestNPFR.js

*Notice: Please run the test case one by one as above instruction, there seems to be an unexplained problem with TestNPFR.js if you run all test with **truffle test**. Sorry for the inconvenience.*

Part 9b: Token test [TestNPT.js]

Test1: Initial test

This test case is used to test total token supply is correct. Also check accounts[0] which is Token contract creator hold all initial tokens.

Test2: Approved transaction

Account0 approve transaction: account0 -> transfer 500 -> account1
Check transaction can be proceeded successful if it has been approved before.

Test3: Unapproved transaction

Account0 un-approves transaction: account0 -> transfer 500 failed -> account1
Check transaction cannot be proceeded successful if it has not been approved before.

Test4: Transfer successful

Check transfer can proceed successfully if there is enough balance. Check balance before and after transaction to make sure transaction is successful.

Test5: Transfer failed

Check transfer cannot proceed if there is insufficient balance.

Part 9c: Fundraising test [TestNPFR.js]

Test1: Initial test

This test case is used to test total token supply is correct. Also check accounts[0] which is Token contract creator hold all initial tokens.

Test2: Bid with Signature test

This test contains following testcase:

[Testcase1]: Investors can bid successfully with valid Signature.

[Testcase2]: Investors cannot bid successful with invalid Signature.

Test3: Bid test

This test is used to check bid function can make bid to contract successfully. Test also checks bid withdraw function can successfully modify bid' s status from true to false.

[Testcase1]: Account 1 make bid successfully

[Testcase2]: Account 2 make bid successfully

[Testcase3]: Account 1 withdraw bid successfully, bid' s status => false

[Testcase4]: Account 2 withdraw bid successfully, bid' s status => false

Test4: Reveal test

This test is used to check reveal function in round2. It contains these testcases.

[Testcase1]: Account 1 reveal bid successfully

[Testcase2]: Account 2 reveal bid failed, because of mismatch nonce

[Testcase3]: Account 3 reveal bid failed, because payment is less than amount need to pay (price * shares).

Test5: Transfer failed

This test use ten accounts to simulate different edge cases which may happen after round2. Contains:

[Testcase1]: Investor' s bid is valid and get deserved shares transfer

[Testcase2]: Investor' s bid is partial valid because this bid makes total share greater than 10000 shares. This investor can get partial shares and partial refund.

[Testcase3]: Investor' s paid more than he should pay can get refunds (actually paid – should paid)

[Testcase4]: Investor' s bid is invalid because total share is greater than 10000, so he can get full refund.

[Testcase5]: Investor' s bid is invalid because he paid less Ether than he should pay for. He can get full refund even his money is not enough.

[Testcase6]: Investor make multiple bids, one is valid, one is invalid. He can get right shares transfer and right refund amount.

[Testcase7]: Investor cannot call issue function because he has never bid before.

[Testcase8]: Investor cannot call issue function multiple times.

[Testcase9]: Beneficiary can get Ether collected during fundraising successfully.

[Testcase10]: Bids can be sort rightfully, bids with same price will maintain order in round1.

```
account:   share:   price:   paid:   valid:   reason:   refund(ETH):
acc1       2000     2       4000   partial  share overflow  2000
acc2       1000     5       5000   yes
acc3       3000     7      21000   yes
acc4       2000     3       7000   yes
acc5       1000     9       9000   yes
acc1       2000     5      10000   yes
acc7       2000     1       2000   no      share overflow  2000
acc8       2000     1       1000   no      share overflow  1000
acc9       500     5        500   no      wrong payment   500
sorted by price: acc5 > acc3 > acc2 > acc1 > acc9 > acc4 > acc1 > acc7 > acc8
valid bids: acc5(1000) > acc3(3000) > acc2(1000) > acc1(2000) > acc4(2000) > acc1(1000)
bid's payment calculated:
           acc5: 1000 * 9 = 9000
           acc3: 3000 * 7 = 21000
           acc2: 1000 * 5 = 5000
           acc4: 2000 * 3 = 6000
           acc1: 1000 * 2 + 2000 * 5 = 12000
Total ETH collected = 53000
```