**COMP6451**

**Assignment 2 – Ethereum Programming**

A picture containing arrow

Description automatically generated

**2022 - Term 1**

**Assignment 2 - Report**

**Kanishka Yamani z5368339**

**Contents**

1. Project Data Model
   1. Interaction Summary
   2. Registry Mapping
   3. Bid Mapping
   4. Bid Linked List
   5. Hashes
2. Off-chain Computation
   1. Investor Logic
3. Meeting Requirements
   1. NeverPay
   2. ASIC
4. Explanation of Functions
   1. NeverPayShares.sol
   2. SICAR.sol
   3. CertificationLogic.js
   4. InvestorLogic.js
5. Stretch Goal – Authorization
   1. Certification Logic
   2. Process
   3. Registry Logic
   4. Sophisticated Investor
6. Running Cost Analysis
   1. ASIC
   2. NeverPay
   3. Investors
7. Ethereum Suitability
   1. Platform Reflection
8. Vulnerabilities
   1. Re-entrancy Attacks
   2. Default Visibilities
   3. Organization Private Key
   4. Investor Private Key
   5. ASIC Registry Management
9. Information Sources
   1. References
   2. Libraries
10. Testing Summary
    1. Testing Approach
    2. Testing Requirements
    3. Testing Summary

**Project Data Model**

Interaction Summary

Within this data model exist four entities which are

* ASIC
* Financial Organizations
* Sophisticated Investors
* NeverPay Company

The interaction between these entities is such that ASIC is an established organization which has created a registry system through which financial organizations may apply in order to become a trusted certification issuer and exist within this registry. Becoming a certification issuer allows this organization to generate legal certificates for investors to prove that they are sophisticated, demonstrated by their abundance of assets or high income. Whereas investors are those who are seeking to support and fund companies to eventually create a profitable investment but must first gain a status of being sophisticated. Such a company is the start-up Fintech NeverPay, which is deciding to organize an equity fundraising to collect funds through the Ethereum platform.

As such, the interaction begins with a regular investor seeking to become a sophisticated investor by contacting their financial organization. This organization will provide the certificate that proves this status, and the investor may now interact with the Ethereum platform to participate in NeverPay’s fundraiser. NeverPay, however, must ensure that this investor has the sophisticated status by interacting with ASIC’s registry, which validates the investor’s certificate. Thus, the investor is eligible to participate and purchase shares.

As financial organizations do not wish to operate on the blockchain, they are not required to interact with other entities, but rather, are only required to provide ASIC with a public key that may be identified by a private key

Registry Mapping

ASIC – Certificate Organization Registry Mapping

The first major element in this data model is ASIC’s registry mapping which exists in its smart contract. This is a mapping of address to string, which represents the public address of the certificate organization, and that value is mapped to the organization’s name. This allows ASIC to manage a registry of addresses of organizations that it chooses to allow the issuance of certificates. A simple O(1) cost-effective method of calling the contract with an address as input will allow the caller to determine whether an address exists within the registry. This process is important for NeverPay’s contract because this is the only viable method through which its smart contract may verify if an investor is certified.

It is important to note that these financial organizations are assumed to have developed their own medium through which they may interact with ASIC in order to gain the status of a certificate issuer and provide documents to ASIC which allows them to add their public key to its registry. This will not be described in detail, in this report.

Bid Mapping

NeverPay – Round One Data Bid Mapping

During the initial bidding process to purchase shares, an investor will provide a blinded bid which will then act as a key for a mapping to a solidity structure ‘Data’, comprising of an address, to identify the investor, and an unsigned integer to record the order in which the bid was placed. This is another crucial element in the overall process, because a mapping allows me to implement a many-to-one relationship, where a single investor, identified by their address, may have many bids.

However, I have chosen to use the bid as the key rather than the investor’s address, because the latter would require a mapping of address (key) to an array of solidity structures which comprise of the bid and the order that it was placed. I have chosen against this solution because maintaining and iterating through an array is gas-intensive, so I found it easier to use the hashed bid as the key, with a requirement for clients to provide unique keys, which do not exist in the mapping. Therefore, this allows an investor to have multiple bids, keeping the data structure as cost-effective as possible, and allowing for an O(1) cost to identify the investor by simply calling the mapping with their bid hash as input.

Bid Linked List

NeverPay – Round Two Data Bid Mapping

Perhaps the most crucial data structure, is the use of a doubly linked list, designed using a mapping of an investor’s bid hash to a solidity structure ‘BidNode’ which, as the name implies, acts as a node class, for this bidding process. Like a traditional doubly linked list, this node will be connected to a node of greater value, and to a node of lesser value. What defines a greater or lesser value is dependent on the bid’s offering of price for each share, and the order in which the bid was placed. This draws a dependency from the roundOneData mapping, to identify this order number. Additionally, the BidNode struct contains data pertaining to the bid including the unsigned integers to represent the offered price per share, the demand of shares, and a deposit, which indicates the investor’s deposited ether during the second round.

It is pivotal to ensure that this doubly linked list remains always sorted, and this requirement makes use of each BidNode’s internal data such as its priceEth and, with a reference to roundOneData, its bid order. Thus, allowing me to recursively sort incoming bids into the linked list by reattaching links. Moreover, following the traditional implementation, a head node and tail node must be identified in order to keep track of the linked list’s size, and to effectively sort while inserting. These are identified by the firstBid and lastBid variables, where the firstBid implies least valuable bid, and the lastBid implies the most valuable bid.

Hashes

Although not strictly a data structure, storage of hashes throughout this system is an important factor to consider, as every entity must deal with the generation, storage, or management of hash values, such as bid hash, or certificate public verification keys, which in Solidity, are represented using bytes32, or bytes of length 65 for the signatures.

**Off-chain Computation**

Investor Logic

Investors, when interacting with the NeverPay contract will be required to produce one-way keccak256 hashes, and make use of an array of hashes, in order to return a specific hash. Located in InvestorLogic.js are the two off-chain helper functions getBidHash() and getBidHint().

The getter function for a bid hash will allow the investor to simply input the amount of shares they wish to purchase, and the price, in ether, they are offering per share and the function will output a structure investorBidInformation that provides information about the bid, including the ether required to cover the cost of the bids, the nonce, and the hash itself, which they may then use in their call to various functions in the contract. It is important for the investor or the medium that the investor uses to keep track of the nonce, as it is cryptographically randomized, and it will be required in order to reveal their bid in the second round.

The getter function to acquire the bid hint is highly crucial in the process of revealing the investor’s bid because its functionality allows the investor to acquire the closest bid, within the linked list of unknown size, that will allow the sorting process to conclude as efficiently and quickly as possible. Without this solution, the investor would be required to find the position in which their bid belongs, by starting from the head or tail node, which is inefficient.

The logic provided in this JavaScript file is expected to be integrated in a front-end system, which thereby, eliminates the need for the investor to install dependencies and tools on their computer in order to compute and execute these functions.

**Meeting Requirements**

NeverPay

Note: Any reference to a line will refer to the NeverPayShares.sol smart contract

**1.** The claimShares() function ensures that the maximum number of shares that may be sold must be 10,000, on line 477. This requirement is only relevant when there is more demand than supply, and if this occurs, then the shares that is supplied is 10,000 – sharesSum, meaning that the totalSupply will be capped at 10,000.

**2.** The purchaseShares() function ensures that the value of the shares that an investor wishes to purchase must be greater than 1, on line 303. Setting price or shares to 0 will not pass this requirement, and both floating point numbers and negative numbers are not accepted as unsigned integer inputs. Therefore, the minimum ether per share is strictly 1 or greater. It is important to note that in my implementation, the investor may not provide decimals at any point, meaning the increments in price paid per share will increase in 1 whole ether. For example, 1,2,3 … n.

**3.** The bid() function takes the investor’s bid hash as an input, and nothing else that evidently identifies an individual with anything other than their Ethereum address, which is stored in the roundOneData mapping, along with their bid hash, which is numbers and letters.

**4.** The bid() and purchaseShares() functions satisfy this, because the auction process begins as a blind auction, due to the obscurity of bid hashes, and the inability to gain information from them directly. Moreover, the bidding round and purchasing round are separated as round 1 and round 2, which acts to satisfy the requirement.

**5.** April 20th, also identifiable as 1650412800, as epoch time, is used in the bid() function to ensure that it may not be called after this round (1) has passed. Moreover, April 27th, or 1651017600 epoch time, is used in purchaseShares() function to ensure that it may not be called after this round (2) has passed. Therefore, actions are rejected after the deadlines.

**6.** When an investor calls the bid() function, to provide their bid hash, they are not providing any other information which allows other investors to identify their true bid information. Keccak256 is a one-way hash which is difficult to decipher in a reasonable time, especially without the random nonce, so it is reasonable to assume that this information is not public.

**7.** The bid() function allows the investor to provide a \_remove parameter which acts as a method of allowing them to add or remove their bids. If 0 is provided, add bid, 1 means withdraw. Moreover, these bid hashes are stored in a mapping, which allows investors to provide multiple bids, as long as they are unique hashes, and they may withdraw and or all of these bids at any given time, during round 1.

**8.** The purchaseShares() function will collect the revealed bid information provided by the investor, including price per share, shares amount, and nonce, and use this information to create another hash, to ensure that their initial bid hash, provided in round 1, matches the data they have now presented. This ensures that investors cannot cheat and mention new values.

**9.** The purchaseShares() function is a payable function which takes the investor’s ether, and the require statement on line 303 ensures that the value of ether provided is equivalent to the shares \* price \* 1 ether, which ensures that the investor provides enough wei to cover the cost of the shares.

**10.** During the call to claimShares(), the function ensures that data regarding the bid hash exists in both roundOneData and roundTwoData, to ensure that the investor has participated in both rounds and has successfully provided enough ether to cover the cost of the shares. Due to this require statement on line 449, investors who do not meet this criteria are unable to successfully claim shares. Moreover, due to the management of the linked list, as it is always sorted as new investors attempt to purchase shares in round 2, the list will always be sorted such that p1 >= p2 >= pn. Hence, the first investor will identify the lowest valid bid, which will then allow this investor and all other investors to check whether their p1 is greater or equal to the lowest valid bid. If so, then they are deemed successful, and may claim shares. Otherwise, they are unsuccessful. For an investor who overpays (this could be an investor who has simply provided more ether than necessary, or the lowest valid bidder who is not eligible for all the shares that they intended to purchase), this investor will have their deposit updated on line 505 – 506, which ensures that they are then refunded whatever they overpaid, on line 523. Moreover, the lowest valid bidder may not receive all the shares they wanted to purchase, and this logic is ensured on line 447, where they receive enough shares to reach the 10,000 shares capacity. Following lowest valid bidder, all other investors are unsuccessful.

**11.** The claimShares() function will consider the investors against the lowest valid bidder by calling the helper function isGreaterThan() which determines if the bid in the first parameter is greater than the bid in the second parameter by comparing price, first and foremost, and then by the order in which the bids were placed in round 1. This logic may be observed on line 352.

**12.** The claimShares() function will allow an investor to refund anything that is owed to them. This applies to unsuccessful investors who call this function and have a deposit that is greater than 0. If so, the requirement on line 518 will be satisfied, and the investor will be refunded what they are owed. A deposit greater than 0 may be acquired by calling the purchaseShares() function, with valid information and providing enough ether to cover the cost of the shares.

**13.** After investor has claimed shares, on line 510, it is noticeable that their balanceOf is increased by the number of shares they purchase. Hence, the investor may call the transfer() function on line 100, to transfer shares by deducting from their balance, and sending these shares to the target address’s balance. The entire ERC-20 interface has also been implemented, as shown using the IERC-20 interface on line 14, and the creation of each of the fundamental functions starting from line 87.

**14.** This smart contract is extremely cost-effective for NeverPay to run because all transaction costs are borne by the investors, rather than NeverPay itself. Moreover, the only cost for the company is the deployment cost of the contract itself. The contract is also highly optimized, as various functionalities are combined, as seen by the bid() function which implements both the placing and withdrawing functionality. Moreover, the purchaseShares() function handles the allocation of shares and the collection of ether from the investors. Finally, the claimShares() function acts as a method of claiming, and a refunding system to any investor to who ether is owed. Many variables are also re-used, and state variables are minimal.

**15.** Due to the time restrictions, I was unable to create a solution that is fair for every investor, cost wise. As can be seen by the cost analysis, provided below, it is only unfair for the first investor who claims shares, as they must find the lowest valid bidder. However, all other investors share almost equal costs for every other operation, which shows that the solution highly prioritises the uniformity of costs imposed on investors. I have used the psychological thought experiment of a train conductor to reach my conclusion on this decision. There is a train, and there are 2 tracks. On one track, there are 5 people, and on another track, there is only 1 person. The train is currently moving directly toward the track with the 5 people. A collision with these individuals will certainly kill them. However, I, as the conductor, can change the direction of the train, so that it may take the other track, which has 1 person standing on it. Now, the question is, do I leave the train as it is, and allow it to kill these 5 people, or change the direction to kill 1 person, and to inevitably save 5 others. This is exactly what my solution does, by allowing a single investor to tank the cost of computing the lowest valid bidder, and the remaining n number of investors are saved, and have drastically lower and uniform costs imposed on them. However, my solution also tracks this investor using the state variable ‘lowestValidFinder’, which enables NeverPay to offer an incentive or a reward to the individual who computes this cost. This reward or incentive does not need to be financial, and can be a psychological one, where they receive a shout-out, or a label / role on media such as Discord, Reddit, or other. They may also be given a contributor role, or some form of praise which may allow the individual to pivot off this opportunity to grow themselves as a character within the community, if they wish. The positive options for NeverPay are virtually endless, and it is up to the company as to how they reward this person.

ASIC

Note: Any reference to a line will refer to the SophisticatedInvestorCertificateAuthorityRegistry.sol smart contract

**1.** As presented on line 26, the contract will store a set of public keys to be used as signature verification keys. These keys are those of financial organisations which have applied to ASIC to become a certificate issuer. Moreover, the use of addKeyToRegistry() and removeKeyFromRegistry() functions allow ASIC to add and remove keys from this registry. Moreover, the require statements in these functions ensure that only ASIC has permission to do so. The ASIC address is stored during contract creation, through the constructor, and the use of the ASIC state variable. There is also the isKeyAuthorized() function which accepts external calls from other smart contracts to search through the mapping to check if a key exists. This will return true if it exists, or false if not.

**2.** The CertificationLogic.js file consists of functionalities which enable certificate authorities to create certificates, using the createCertificate() function. Moreover, the applicationForm() function enables investors to apply to this financial organization to apply for the certificate. This must be implemented through a user-friendly front-end.

**3.** The bid() function uses the function caller’s address and the current year to create a signed eth message, which the function then uses, along with the public key signature that the investor provides, to ensure that the investor has a sophisticated investor status. This is done through the ecrecover function which allows the combination of the signature and the signed message to retrieve the message signer, which is then sent to the ASIC smart contract to check whether this signer exists in its registry. If it exists, then the investor is verified. If not, then the investor cannot place a bid.

**Explanation of Functions**

NeverPayShares.sol

**transfer()** – Allows an investor to transfer their shares to another person's address.

**approve()** – Enables an investor to give an allowance of the shares they own, to another address.

**transferFrom()** – If a person has approval for an allowance of shares, they may transfer them on behalf of the investor.

**bid()** – Enables a sophisticated investor to place or withdraw a blinded bid for a price to buy a number of shares. The place and withdraw functionality is encapsulated in this function to reduce deployment cost.

**purchaseShares()** – Investor may purchase shares during round 2, while revealing their initial blinded bid information.

**isBidGreater()** – Helper function to check whether the bid in the first parameter is more valuable than the second bid parameter.

**bidPlacement()** – Recursively changes the order of a given bid to ensure that it is sorted (smallest -> largest).

**claimShares()** – Allows a successful investor to claim the shares they have purchased in the previous round and claim a refund if eligible. The first investor will do the heavy computation, but their address is identifiable by the contract, to allow NeverPay to offer incentive.

SophisticatedInvestorCertificateAuthorityRegistry.sol

**isKeyAuthorized()** – Checks if the provided key exists in the authorized organization registry

**addKeyToRegistry()** – Adds the provided key and business name to the registry mapping

**removeKeyFromRegistry()** – Removes an organization from the registry (may no longer verify investors)

CertificationLogic.js

**applicationForm()** – The application form functionality will be represented through a front-end online application form that the investor, wishing to verify themselves as a sophisticated investor, must complete successfully. The organization may provide instructions, code, or a software to enable investors to "Securely" generate a signature

**createCertificate()** – The certificate creation process is performed in the backend of the organization, because it consists of private information pertaining to real-life people, and the private key of the organization, which is highly susceptible to being compromised and misused. The parameter is the investor's public address. However, this function is responsible for creating the certification for the investor to use as proof of sophisticated status.

**createCustomCertificate()** – The following code is not strictly used for certification or the application form but complements the truffle testing process. It is used to generate an X number of financial organizations which can sign messages, and allow investors to become sophisticated investors, if their public wallet address is provided as a parameter.

InvestorLogic.js

Note: Any reference to a line will refer to the SophisticatedInvestorCertificateAuthorityRegistry.sol smart contract

**getBidHash()** – This function will allow an investor to combine the amount of shares they're willing to purchase and the price they wish to pay per share (in ether) into a packed keccak256 (sha-3 family) one-way hash. The nature and security that this hash provides will ensure that the investor's bid information will remain anonymous to an utmost extent. Moreover, a nonce, produced using crypto.randomBytes(32), will allow the investor's bid information to become much more randomized, as this nonce is cryptographically well-built artificial random data. This nonce will need to be stored by the investor and what I mean by this is the off-chain application through which they are interacting with the contract. In this way, all of this information maybe reliably saved, and the investor may provide this information during round 2, when they are purchasing shares, making the validation process easier.

**getBidHint()** – In order to get a hint, a bid must be given as a parameter to this function so it can be compared against all existing bids in the contract, if any. The parameters in the getBidHint function will be bidHash and currentBid. The bidHash variable may be obtained by calling the getBidHash function by passing the investor's bid information such as number of shares, and price per share. The currentBid variable may be obtained by calling NeverPayShares.sol smart contract and retrieving the value of public variable firstBid. This value is passed as the initial comparison. This function does not break trust between the investor and the client (NeverPay) because it only serves to reduce the computation time required to find the location to insert a new bid. And this is a vital functionality to ensure that the bids list remains always sorted from index = 0 being the lowest valued bid, and index = n being the highest valued bid. If an investor were to provide a false bidHint, then the transaction itself would potentially cost more gas, as it may take longer to find the correct position to insert the investor's bid, to ensure that the list remains sorted.

**isBidGreater()** – A helper function, like the one which exists in the NeverPayShares contract, to identify the higher bid. Determines whether the first bid (order, price) is greater than the second bid (secondOrder, secondPrice). If first bid is greater, return true, otherwise return false to imply that the first bid is less valuable.

**Stretch Goal - Authorization**

Certification Logic

To implement the stretch goal, a JavaScript file, CertificationLogic.js was required in order to allow financial organizations to produce a certificate using the investor’s Ethereum address, and then provide this certificate through e-mail to the investor. This investor may then use this certificate to prove their status as sophisticated.

Moreover, an application form process was implemented in this logic, to demonstrate how the investor would interact with the financial organization front-end in order to come to undergo the process of acquiring this certificate.

Certificate Organizations must be financial firms which have access to the investor's personal details, documents, financial history, and must be aware of their net worth (assets / annual income).

Process

> ASIC will ensure that any financial firm that wants to issue certificates is trustworthy and undergoes a rigorous application process, supervised by ASIC, to be approved as a trusted organization.

> The financial information that investors submit to these trusted organizations must be legitimate, and it is the duty of the financial firm, and the Australian Law to ensure that fraud is not tolerated.

Involved Parties:

=================

> Certificate Organization

> Investor

> ASIC

Process of issuing a certificate:

=================================

Phase\_1:

========

> Investor operates through organization, so they will visit organization's online website and login.

> Their goal is to complete the online application form to gain a sophisticated investor certification.

> The organization has access to most of their personal details, identity documents, and financial history.

> Investor's net worth is also known, through their owned assets and annual income, as per financial statements.

> There will be a section in the application form which pertains to verifying cryptographic proof.

> The investor will be instructed to sign a message, given by the form, using their crypto private key.

> It is important to note that they will do this locally on their machine, to keep their private key secure

> Investor will input their public signature verification key and crypto address (hashed public key) in the form.

> Organization ensures that investor meets the financial criteria for sophisticated investor and proceeds to Phase\_2.

Phase\_2:

========

> Organization will collect the information provided by investor and pass as parameters to 'checkInvestorValidity'

> If the 'checkInvestorValidity' function returns true, then ownership of the provided address is verified.

> A certificate will now start to be created to act as a proof that they are a sophisticated investor.

> Message: "the owner of Ethereum address <address> is a sophisticated investor for year <year>."

> The address and year fields will be populated by their verified crypto address and the current year (2022 now).

> Organization will have its own private and public signature keys available in its database, for signing purpose.

> Message is hashed as Ethereum Signed Message and signed by organization's private key to produce a public signature.

> The application process will end after organization sends an email to investor about their verification information.

> This verification info includes a copy of the signed message, and organization's public signature verification key.

> Investor provides this info, when they place a bid for NeverPay shares, to prove that they are a sophisticated investor

It does not compromise the investor because they do not need to give out their private key to complete this verification process. Moreover, it is not reasonably possible to compromise this process because a malicious user would still require access to a private key in order to impersonate another person, due to the encryption requiring the private key, and decryption requiring the public key.

Registry Logic

To implement the registry logic, the SophisticatedInvestorCertificateAuthorityRegistry.sol was required to be created which stores the organisations’ keys, and allows external contract calls to verify if the address they provide exists within the registry. ASIC may also add and remove keys as they deem fit. This satisfies all the requirements laid out in the spec.

Sophisticated Investor

The verification process to ensure that an investor has a sophisticated status is then completed in the bid() function in the NeverPayShares.sol contract. Using the signature that the investor provides and then creating an eth signed message, using the investor’s Ethereum address, along with the current year (2022), their status is verified by calling the ASIC contract.

**Running Cost Analysis**

ASIC

Text

Description automatically generated

The cost of deploying the smart contract for ASIC would be 572,916 units of gas.

NeverPay

Text

Description automatically generated

I have used: <https://etherscan.io/gastracker> - to understand average gas prices

I have observed gas prices as low as 18 gwei, and as high as 32. Currently, on Monday, 11th April 8:28pm, the gas price is 23 gwei. However, I will use a slightly higher price, to account for congestion, such as 25 gwei to carry out the analysis using this as the gas price.  
  
To edit the gas price, simply change --gasPrice=<1000000000 \* gas price (gwei)> when using the ganache client. It's useful to do this when you carry out your test to get an approximation of the deployment cost at the current, local time.

CostEth = 0.0538988  
gasCost = 2155952

Using <https://www.coinbase.com/converter/eth/usd>, as of 11:13pm 11th April, with a conversion rate of ether : USD => 1 | 3051.14, the gas price is: $164.45 USD.

In terms of the running cost for NeverPay, the only cost the company will endure is the cost of deploying the contract. This will always cost a constant value of 2148360 gas. The code is highly optimized, so as to reduce costs as much as possible (talk about how I joined functions together, re-used variables wherever necessary, minimised the use of state variables, used appropriate function names private/external/etc.., used variables to store values instead of referring to state variables each time, initialized variables with default value (e.g., uint x;) to reduce the cost). Moreover, it is still easily readable, and with the use of annotations throughout the smart contract, it's easy to understand what each function and variable is responsible for. From NeverPay's perspective, the price of 1 ether is over $3051 USD, and the cost of deployment is a mere $164. This also means that the purchase of a single bid will recoup NeverPay's lifetime cost by over 18 times.  
  
After which, NeverPay will not endure any other running cost (because all transaction costs are borne by investors), and subsequent bids will bring forth full profit for the company. The only area of slight concern for NeverPay is giving an incentive for the first person to call the claimShares function because this investor has to undergo extra computation. However, my proposal is that NeverPay doesn't have to give the investor a financial incentive, but rather a psychological one, where they may receive a contributor role within a community, such as Discord, or Reddit (social media). Or perhaps they get some form of slight recognition by creating and giving them a virtual label or shoutout. An investor may be highly incentivised by this, as they can use this platform to build their own image within the community or reach a wider audience with their own ideas/projects. There are several options for what NeverPay can do, which does not involve money.

Investors

Cost of placing a bid:  
> 50 investors  
> Around 82k

Graphical user interface

Description automatically generated

Cost of withdrawing a bid:  
> 50 investors  
> Around 25k

Graphical user interface

Description automatically generated

Cost of purchasing shares:  
> 50 investors  
> Around 157k

Graphical user interface, text

Description automatically generated with medium confidence

Cost of claiming shares:  
> 50 investors  
> first investor cost: 55737 (cost to claim shares) + 62315 (cost of iteration) + 3809 \* (n - 1), n > 0 (where n = number of investors who purchased shares during round 2)  
> All other investors: around 55k

Calendar

Description automatically generated with medium confidence

With a block limit of 30 million gas: <https://ethereum.org/en/developers/docs/gas/>

EIP-1599 (London Hard Fork) brought variable-size blocks to the Ethereum network. This allows a gas limit of 30 million.

118052 = 55737 (cost to claim shares) + 62315 (cost of iteration) + 3809 \* (n - 1), where n = 1

In order to compute the lowestValidBid when there are 10,000 investors, it would cost 118052 + 3809 \* (10000 - 1) = 38204243 gas. This is clearly over the limit.

The max number of investors, for the gas to reach 30 million is ((30000000 - 118052)/3809) - 1 = 7844 investors. (I got this by rearranging the cost for finding the lowest valid bid)

This means that the smart contract is limited to having a total of 7844 investors. Any more than this and the block limit would reach, and the transaction would not go through.

One possible solution I had, to counter this issue, is to have a temporaryLowestValidBid state variable along with a method of detecting if the number of iterations reached close to the maximum number of investors that the function can iterate through, before the limit is reached. So, this means that if the while loop iterates, say, 7800 times, then this 7800th investor bid hash would be stored in temporaryLowestValidBid. Once another investor calls the function, it would resume the loop from this point onwards, till the 10,000th investor is found, and the lowestValidBid value is set. This would require modifying the function by

a) Checking if the function was previously left in a state where the lowestValidBid has not been found yet, therefore iterate from the 7800th point onwards, instead from 1st investor onwards.

b) Collecting the hashes of the two investors who are responsible for finding the lowestValidBid, in order to give them both incentives / rewards for doing so.

c) Creating state variables to store these hashes inside. Then NeverPay would have to also increase the amount of effort they put into this incentive system, marginally.

There would be two investors who find the lowestValidBid, if there were 10000 investors, because 7800 \* 2 > 10000, so by the time that the function is called by two investors, the lowestValidBid is found. The reason that I did not implement this solution is because I simply did not have enough time to create the code and test it to ensure that it works accordingly. However, the solution I have in place currently is a close alternative which does the job of distributing shares well, until the point that there are more than 7844 investors in the linked list.

**Ethereum Suitability**

Platform Reflection

Ethereum has become a very congested platform due to the recent uprise of NFT, play-to-earns, and many popular Meta and DeFi projects launching on the platform. The congestion is caused by the growing number of people who are flocking to the platform in order to participate in the activities of these projects. There is also the major issue of gas wars, and gas prices skyrocketing during periods of time where projects launch their product and attempt to collect funds. This causes the investors for various projects to compete and raise the gas price to a level which may not be as profitable, or quite costly.

However, I think that this platform is suitable for the company because it is a platform that already has a lot of popularity, which allows a start-up company to quickly gain popularity, and because the Ethereum community is large, there is a rather high chance that the company will find more suitable investors and subsequently, more demand for people that want to purchase their shares.

The platform is currently plagued by a flocking mentality, where if one investor finds interest in a project and proceeds to advertise it, then many others will also participate in the project. If NeverPay wishes to quickly and reliably sell all their shares, then the Ethereum platform is a viable choice. However, it will need to ensure that it maintains a level of standard so that its investors are willing to invest. Due to the high price of 1 Eth (over $3000 USD), and having 10,000 shares available, almost like a 10,000 NFT project, the shares purchasing process may experience a slight gas war, so this is something NeverPay will have to keep in mind in order to regulate the fundraising process and allow investors to pay more uniformly.

**Vulnerabilities**

Re-entrancy attacks

There is a popular ‘hack’ that users may perform in order to call functions multiple times before they have finished executing. I have taken note of this and have created a modifier in the NeverPayShares.sol contract called ‘reEntrancyBlock’ which ensures that the purchaseShares() and claimShares() functions are not susceptible to this exploit. It will basically set ‘blocked’ to true and disallow function execution until the blocked value is set to false again, which happens after the function finishes executing.

Default Visibilities

Some of the helper functions may be abused in the smart contract such as bidPlacement() and isBidGreater() which is no longer possible because the visibility for those is set to private, allowing only the contract itself to call these functions.

Private Keys

During the certificate creation process, it is important that the investor and the financial organization does not reveal its private keys to anyone. This process of signing messages must be done locally to ensure that no breach may occur.

ASIC Registry Management

It is important to consider who and how ASIC manages its registry. Allowing the private key of the ASIC address to be leaked or compromised is a very large breach which affects any external contract which uses its functionality, so it is important to store this in a backend, which is not easily accessible or somewhere more private.

**Information Sources**

References

Blind Auction idea:  
https://docs.soliditylang.org/en/v0.8.13/solidity-by-example.html (operating under GPL -3.0)

Idea of using nonce:  
<https://stackoverflow.com/questions/70566188/node-js-crypto-randombytes-is-not-a-function>

Thought to implement a linked list generated from this reference:  
<https://medium.com/coinmonks/linked-lists-in-solidity-cfd967af389b>

Understanding the ERC-20 standard:  
<https://www.investopedia.com/tech/why-crypto-users-need-know-about-erc20-token-standard/#:~:text=Key%20Takeaways,standard%20was%20implemented%20in%202015>.

Used this to help setup testing environment (Truffle + Ganache / testnet):  
<https://www.youtube.com/watch?v=YYJgeV7sOvM&t=1263s>

for signing /verifying messages, I used these as guides:  
<https://www.youtube.com/watch?v=vYwYe-Gv_XI>  
<https://yos.io/2018/11/16/ethereum-signatures/>

documentation to sign messages:  
<https://web3js.readthedocs.io/en/v1.7.1/web3-eth-accounts.html#sign>

Libraries which must be installed

1. npm install –save ethers
2. install node.js and npm
3. install truffle
4. npm install truffle-assertions
5. npm I ganache-time-traveler
6. I remoed node\_modules from the assignment folder, please install the libraries which are used within the js files, if not explicitly mentioned.

**Information Sources**

Testing Approach

When I write tests, I ensure that I only test for one thing per test. So, if I test, for example, if an investor is purchasing over 100 shares, then I will assume that they are a valid bidder, and passed all the other requirements to reach this stage of the testing process. I do this to ensure that I’m only focusing on one instance of a test at a time. I will test the latter assumptions in various other tests. This will result in many tests having similar code, but the area of focus will be different, and this will be reflected by the documentation I design for the tests (in this report, explaining what part of the logic that each test is meant to ensure works properly) and also based on the assert error messages, and the test title, which represents this area of focus.

When I use decimals in the tests, I need to abi.encode / pack these values as strings, because solidity deals with whole number integers.

test each of the test files separately, and don’t run ‘truffle test’ because I am using the ganache-time-traveller which is somewhat finnicky and riddled with ways in which all tests or some tests could fail due to unknown reasons. I have done a lot of research, but I found that testing each file one-at-a-time is the most reliable way to verify that all tests do in fact pass.

Testing Requirements

1. Open terminal, set directory to NeverPay

2. run the following:  
$ set NODE\_OPTIONS=--openssl-legacy-provider

3. Start the ganache client:  
$ ganache-cli -e 20000 --time 04/04/2022 --gasPrice=25000000000 --gasLimit=6721975 -a 20

4. Open another terminal, set directory to NeverPay

5. To run each test, use:  
truffle test .\test\<contractFolder>\<testFolder>\<testFile>

**Example:**  
truffle test .\test\1\_NeverPayShares\1.2\_SharePrice\1.2.3\_CanBuySharesIfAtleast1Ether.test.js

5.1. (Alternatively) You may set directory to the test folder, and call directly, example:   
truffle test 1.2.3\_CanBuySharesIfAtleast1Ether.test.js

Testing Summary

**Test** **-** **1.1.1\_Max10kShares.test**  
 **🡺** **What it tests:**

* *Test that a maximum of 10,000 shares can be sold in total (after round 2)*

**🡺** **Importance:**

* NeverPay intends to sell 10,000 shares in total, throughout this bidding process. Therefore, this test ensures that, if there were more demand, no more than this number may be claimed, as doing so would severely go against NeverPay’s requirement (Notably, spec requirement .1).

**🡺** **How it works:**

* This test uses 10 investors, marked by totalInvestors = 10, and loops through the bidding and purchasing process 10 times. Each investor will purchase 1,100 shares, therefore, 10 \* 1100 = 11,000. This is greater than 10,000, and therefore, the assert.equal() at line 94 tests whether the totalSupply remains at 10,000, as this variable indicates the number of claimed shares. If so, then despite the high demand, the contract successfully sold a maximum of 10,000 shares.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.1\_Shares\*1.1.1\_Max10kShares.test.js*

**Test** **-** **1.1.2\_OneManCantBuyMoreThan10k.test**  
 **🡺** **What it tests:**

* *Test that a single investor can’t buy more than 10,000 shares after round 2*

**🡺** **Importance:**

* NeverPay intends to sell a maximum of 10,000 shares, and so, if the demand, from a single investor, is over this number, then they will be limited to purchasing the entire supply of 10,000, rather than the amount they bid for. This is pivotal to satisfy spec requirement .1.

**🡺** **How it works:**

* Only a single investor has placed a bid and purchased shares in this iteration of the auction process. However, they have offered to purchase 14,001 shares for 1 ether each. As the maximum supply is 10,000 shares, after calling claimShares() at line 90, this test would need to ensure that the investor would still only have 10,000 shares, noticeable at line 92, as their balance is equal to 10,000. Moreover, the totalSupply state variable also detects, at line 96, that there have only been 10,000 shares which have been claimed.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.1\_Shares\*1.1.2\_OneManCantBuyMoreThan10k.test.js*

**Test** **-** **1.2.1\_CantBuyForLessThan1Ether.test**  
 **🡺** **What it tests:**

* *Test that investors can’t buy shares for less than 1 Ether per share*

**🡺** **Importance:**

* NeverPay wishes to sell its shares for a minimum of 1 Ether per share, as noted by spec requirement .2. As such, it is important to test that an investor is unable to purchase a share for less than this amount per share.

**🡺** **How it works:**

* Only a single investor has placed a bid during round 1. Due to the parameters that purchaseShares() solidity function accepts, (uint to be specific), it may not receive floating point decimal numbers as input. Therefore, the test ensures that Ganache will revert the transaction with a NUMERIC\_FAULT error, as demonstrated on lines 46 -47. Then, at line 50, the test will fail if Ganache reports this incident. If, however, this is not detected, then there is a fault with the smart contract, as it accepts decimals as input.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.2\_SharePrice\*1.2.1\_CantBuyForLessThan1Ether.test.js*

**Test** **-** **1.2.2\_CantBuy0Shares.test**  
 **🡺** **What it tests:**

* *Test that investors can’t buy 0 shares during round 2*

**🡺** **Importance:**

* It would be impractical to allow an investor to undergo the auction process if they intend to purchase 0 shares, as this brings no profit for NeverPay, and simply acts to extend the list of investors, and unnecessarily increase competition. Therefore, this test ensures that a minimum of 1 share must be bought.

**🡺** **How it works:**

* Only a single investor has placed a bid during round 1. During this purchasing process, the contract must ensure that the value of shares \* price > 0, meaning that the investor is willing to purchase at least 1 share for a minimum of 1 ether. Otherwise, at line 44, the truffleAssert.reverts() function will ensure that the contract will revert the transaction, as 0 shares are not purchasable.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.2\_SharePrice\*1.2.2\_CantBuy0Shares.test.js*

**Test** **-** **1.2.3\_CanBuySharesIfAtleast1Ether.test**  
 **🡺** **What it tests:**

* *Test that investors can buy shares if they provide more than 1 ether per share*

**🡺** **Importance:**

* This test ensures that a valid bid, where the number of purchasing shares > 0 and the price offered per share > 0, enables an investor to successfully purchase shares during round 2. If this functionality were not implemented, then investors would not be able to purchase shares.

**🡺** **How it works:**

* An investor has placed a bid to purchase shares. During round 2, they reveal that their offer is to purchase 1 share for 1 ether, which is a valid bid, as per NeverPay’s requirements. Therefore, at line 43, the test ensures that the contract call to purchase said shares is not reverted, and if so, the investor has successfully purchased shares and deposited ether.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.2\_SharePrice\*1.2.3\_CanBuySharesIfAtleast1Ether.test.js*

**Test** **-** **1.3.1\_CheckSystemStoresData.test**  
 **🡺** **What it tests:**

* *Test that the system stores the data of the investor (linked to the bid hash in the map)*

**🡺** **Importance:**

* It is pivotal to ensure that the investor’s initial blinded bid, during round 1, is stored in the contract, because this is the only means through which their participance in this auction may be verified during later rounds. It is also essential, as it allows the contract to identify the user through their address, and also the order in which their bid was placed.

**🡺** **How it works:**

* When the investor places a bid, their data is stored in roundOneData mapping. This test ensures that after the contract call, that the bid hash they provided as input to the bid() function will be stored in the mapping and will not have default values. This is noticeable on lines 25-26, where the test checks that the bid information is not equal to default values of 0, implying that the investor’s data is saved.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.3\_ Anonymous\*1.3.1\_CheckSystemStoresData.test.js*

**Test** **-** **1.3.2\_InvestorAddressLinkedToBid.test**  
 **🡺** **What it tests:**

* *Test that NeverPay maps the investor’s bid tot heir address, to track their bids*

**🡺** **Importance:**

* It is crucial to save an investor’s information during round 1, however, it is even more important to ensure that the correct data is being saved. Because, if incorrect data is saved, then the bid owner is unable to be identified during later rounds. It’s important to note that the relevant data that the investor passes is their bid hash, which does not reveal their identity and keeps them anonymous.

**🡺** **How it works:**

* When the investor places a bid, their data is expected to be saved. This is tested on line 25, where roundOneData is checked to see if a bid hash exists that resembles the investor’s intended bid, and that the mapped address in the struct is the investor’s address. If this is the case, then the investor’s identity is stored, and may be used during later rounds to verify ownership of the bid.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.3\_Anonymous\*1.3.2\_InvestorAddressLinkedToBid.test.js*

**Test** **-** **1.4.1\_PlaceABidForLessThan1Ether.test**  
 **🡺** **What it tests:**

* *Test that NeverPay cannot identify bid information due to keccak256 one-way hashing*

**🡺** **Importance:**

* Due to the blinded nature of this auction, it is important to ensure that the bid information is not publicly accessible/viewable. This may be done through one-way hashing, which proves incredibly difficult to decrypt without knowing the parameters. The use of a nonce, during the bid hash generation process ensures that the investor’s bid is ever-the-more random, and harder to reveal.

**🡺** **How it works:**

* This is tested by allowing the investor to place a bid using a value of < 0 for their shares and the price per share, as demonstrated on line 18. This is important because after the hashing process, the bid’s shares/price information cannot be detected by the contract, or viewable by others investor until the hash is verified using the values through which the hash was formed. On line 25, if the contract stores the bid data, despite placing an invalid bid (shares < 0), then it proves that the bid information is truly blind.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.4\_Spec\_Blind\*1.4.1\_PlaceABidForLessThan1Ether.test.js*

**Test** **-** **1.5.1\_BidFunctionCanBeCalledRound1.test**  
 **🡺** **What it tests:**

* *Test that a blinded bid can be placed during round 1*

**🡺** **Importance:**

* NeverPay intends to sell its shares through a round-by-round process, and so round 1 is dedicated for placing bids. It is crucial to ensure that bids may be placed during this round, if valid, in order to process them during later rounds.

**🡺** **How it works:**

* This is tested by allowing a single investor to place a bid using valid bid information and the correct function parameters, and to ensure that the contract call transaction does not revert. If it does revert, on line 23, then bids may not be placed during round 1.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.5\_Deadlines\*1.5.1\_BidFunctionCanBeCalledRound1.test.js*

**Test** **-** **1.5.2\_BidFunctionCannotBeCalledAfterRound1.test**  
 **🡺** **What it tests:**

* *Test that a blinded bid cannot be placed after round 1*

**🡺** **Importance:**

* NeverPay does not wish to receive anymore bids after round 1 deadline has passed, as specified by requirement spec .5. Hence, it is important to ensure that bids may not be placed after this timeframe has passed.

**🡺** **How it works:**

* This is tested by allowing a single investor to place a bid using valid bid information and the correct function parameters after time has passed. The time is changed to reflect that round 1 has ended. Now, if the investor wishes to place this valid bid, then it is expected, at line 38, that the transaction will be reverted. If this is not the case, then investors may place bids after round 1 deadline, which is not intended.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.5\_Deadlines\*1.5.2\_BidFunctionCannotBeCalledAfterRound1.test.js*

**Test** **-** **1.5.3\_BidFunctionCannotBeCalledAfterRound2.test**  
 **🡺** **What it tests:**

* *Test that a blinded bid cannot be placed after round 2*

**🡺** **Importance:**

* NeverPay does not wish to receive anymore bids after round 1 deadline has passed, as specified by requirement spec .5. Hence, it is important to ensure that bids may not be placed after this timeframe has passed. Moreover, bids may not be placed after round 2 either.

**🡺** **How it works:**

* This is tested by allowing a single investor to place a bid using valid bid information and the correct function parameters after time has passed. The time is changed to reflect that round 2 has ended. Now, if the investor wishes to place this valid bid, then it is expected, at line 38, that the transaction will be reverted. If this is not the case, then investors may place bids after round 2 deadline, which is not intended.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.5\_Deadlines\*1.5.3\_BidFunctionCannotBeCalledAfterRound2.test.js*

**Test** **-** **1.5.4\_CantBuySharesInRound1.test**  
 **🡺** **What it tests:**

* *Test that shares cannot be purchased in round 1*

**🡺** **Importance:**

* NeverPay does not intend to sell shares in round 1, but instead, only accept bids in this round. Therefore, it is important to ensure that shares cannot be purchased until the appropriate time.

**🡺** **How it works:**

* This is tested by allowing a single investor to place a bid during round 1, and before the time for round 1 deadline has passed, the investor attempts to purchase their shares, and the transaction should revert, as the time for round 2 has not arrived. If this function call does not revert, then shares may be purchased before round 2, which is not intended.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.5\_Deadlines\*1.5.4\_CantBuySharesInRound1.test.js*

**Test** **-** **1.5.5\_CanBuySharesDuringRound2.test**  
 **🡺** **What it tests:**

* *Test that shares can be purchased during round 2*

**🡺** **Importance:**

* NeverPay wishes to sell its shares during the round 2 period. Hence it is vital to ensure that this functionality is implemented in the smart contract.

**🡺** **How it works:**

* This is tested by allowing a single investor to place a bid during round 1, and after enough time has passed for round 1 to end, and round 2 to begin, the investor then attempts to purchase shares. If the transaction is reverted, then the contract is unable to accept purchase requests for shares. However, at line 44, if the transaction is not reverted, then the contract works as intended, as shares are purchasable in round 2.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.5\_Deadlines\*1.5.5\_CanBuySharesDuringRound2.test.js*

**Test** **-** **1.5.6\_CantBuySharesAfterRound2.test**  
 **🡺** **What it tests:**

* *Test that shares cannot be purchased after round 2*

**🡺** **Importance:**

* NeverPay only wants to sell their shares in round 2, not before or after this time frame. Hence, a test must be made to ensure that the contract functions as such.

**🡺** **How it works:**

* This is tested by allowing a single investor to place a valid bid during round 1. The investor does not perform any actions during round 2. However, after round 2 deadline has passed, they proceed to call purchaseShares() and the test must ensure that the transaction is reverted, as it implies that the time for round 2 is over, and hence, the investor may no longer purchase shares.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.5\_Deadlines\*1.5.6\_CantBuySharesAfterRound2.test.js*

**Test** **-** **1.6.1\_EnsureBidInfoIsReallyBlind.test**  
 **🡺** **What it tests:**

* *Test that the bid does not publicly reveal the investor’s shares or price per share*

**🡺** **Importance:**

* NeverPay wishes to operate through a blinded bid system, so it is of utmost importance that other investors are not able to visibly see other investors’ bid information in a reasonable time/effort.

**🡺** **How it works:**

* The bid function accepts a bid hash, which is an accumulation of the bid itself, such as the shares, price, and an attached nonce. This test will ensure that an investor who is purchasing 25 shares for 4 ether each is not identifiable directly based off this hash. As seen on line 29 and 30, the bid hash must not equal either 25 or 4, which indicate their bid information. If this is the case, then the bid is blind, as the other investors would need to decrypt the hash in order to find out their bid information.

**Requirements:** *default requirements*  
**Terminal:** truffle test .\test\1\_NeverPayShares\1.6\_Round1\*1.6.1\_EnsureBidInfoIsReallyBlind.test.js*

**Test** **-** **1.7.1\_CorrectParameterToPlaceWithdrawBids.test**  
 **🡺** **What it tests:**

* *Test that a bid may be placed when passing 0 and withdrawn when passing 1 as \_remove value*

**🡺** **Importance:**

* The NeverPay smart contract holds both the place and withdraw functionality within the single function, and so based on the value of \_remove, it will dictate whether to add or remove a bid.

**🡺** **How it works:**

* A single investor will place a valid bid by inputting 0 as \_remove value, on line 23. After which, they will remove this bid by using 1 as \_remove value, on line 24. It is expected that the transaction calls do not revert or produce an error, because the bids are valid and this is the intended functionality, where 0 = place bid, and 1 = remove bid. If it reverts, then the contract is not coded appropriately.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.7\_MultipleWithdraw\1.7.1\_CorrectParameterToPlaceWithdrawBids.test.js*

**Test** **-** **1.7.2\_IncorrectParameterValueDoesNothing.test**  
 **🡺** **What it tests:**

* *Test that a bid may not be placed or withdrawn if the \_remove parameter is not 0 or 1*

**🡺** **Importance:**

* The NeverPay smart contract holds both the place and withdraw functionality within the single function, and so based on the value of \_remove, it will dictate whether to add or remove a bid. If this value is not either of these numbers, then the function will not allow the investor to place or remove a bid.

**🡺** **How it works:**

* A single investor will place a valid bid by attempting to input 999 as the value of \_remove. It is expected that the contract does not save the bidder’s information because a value of 0 is required as \_remove, in order to place bids. This is tested on lines 25 – 26, where the test assert.equal() that the bid Hash does not exist within the roundOneData mapping, by ensuring that the values within the struct are default (0).

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.7\_MultipleWithdraw\1.7.2\_IncorrectParameterValueDoesNothing.test.js*

**Test** **-** **1.7.3\_OrderCannotBe0.test**  
 **🡺** **What it tests:**

* *Test that a bid, which is not withdrawn, cannot have an order of 0*

**🡺** **Importance:**

* The contract is intended to function such that the bid’s order is always 1 or greater, if their bid is valid, because the order resembles the order in which bids were placed, from 1 being the first order and n being the last order. If the order, however, is 0, then it implies that the bid is no longer viable (withdrawn).

**🡺** **How it works:**

* A single investor will place a valid bid. The test must ensure, then, that the order value within the struct for roundOneData is not equal to 0, as demonstrated on line 25, assert.notEqual() will ensure that the bid was not withdrawn, because the bid function was not called with a \_remove value of 1.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.7\_MultipleWithdraw\1.7.3\_OrderCannotBe0.test.js*

**Test** **-** **1.7.4\_InvestorCanPlaceMoreThan1Bid.test**  
 **🡺** **What it tests:**

* *Test that a single investor may place multiple bids, and this data is saved*

**🡺** **Importance:**

* NeverPay has intended for investors to be able to place multiple bids, as investors may collect shares from multiple bids, after purchasing shares.

**🡺** **How it works:**

* This is tested by allowing an investor to place 5 bids, represented by the totalBids value. After 5 iterations, during each iteration, the test will ensure that the bid exists within the roundOneData mapping, by ensuring that the values within the struct do not have default value (0), as shown on lines 30 – 31.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.7\_MultipleWithdraw\1.7.4\_InvestorCanPlaceMoreThan1Bid.test.js*

**Test** **-** **1.7.5\_CanPlaceAndWithdrawSingleBid.test**  
 **🡺** **What it tests:**

* *Test that an investor may place a single bid and withdraw this bid*

**🡺** **Importance:**

* NeverPay has intended for investors to be able to place a bid, and also enable the investor to withdraw the bid, if they no longer wish to purchase shares.

**🡺** **How it works:**

* A single investor will begin by placing a valid bid, using 0 as \_remove value. This is checked at lines 26 – 27 by ensuring that the roundOneData struct does not have default values (0). After which, the investor will withdraw this bid by inputting the same bid Hash and providing 1 as \_remove value. Following this, the test will ensure that the id of the investor (stored address) does not change, but instead, the order becomes 0, as shown by line 32. An order of 0 implies that a bid is withdrawn.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.7\_MultipleWithdraw\1.7.5\_CanPlaceAndWithdrawSingleBid.test.js*

**Test** **-** **1.7.6\_CanPlaceAndWithdrawMultipleBids.test**  
 **🡺** **What it tests:**

* *Test that an investor may place multiple bids, and withdraw multiple bids*

**🡺** **Importance:**

* NeverPay has intended for investors to be able to place multiple bids and allows the investor to withdraw multiple bids. However, the real importance of this test is to ensure that adding and removing bids, at ‘scale’, works as intended.

**🡺** **How it works:**

* A single investor will begin by placing 5 valid bids, as represented by totalBids = 5. Following this, they will withdraw all 5 bids. The test will ensure that bids are placed by checking that their id and order is stored, on lines 31 – 32. Moreover, it will ensure that all bids are removed by checking that the order = 0, on line 43. The bid hashes themselves are stored in a bids[] array, which allows to keep track.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.7\_MultipleWithdraw\1.7.6\_CanPlaceAndWithdrawMultipleBids.test.js*

**Test** **-** **1.7.7\_NoDuplicateBidsAreAllowed.test**  
 **🡺** **What it tests:**

* *Test that placing a duplicate bid does not add to the bids mapping, and does nothing*

**🡺** **Importance:**

* The contract should not accept duplicate bids, as the bids themselves, are stored in a mapping, where the bid hash itself is used as the key. So, allowing an investor to place duplicate bids would essentially replace the values within the struct with a new set of values. Hence this functionality should not be allowed.

**🡺** **How it works:**

* A single investor makes a valid bid on line 26. It is verified by the test that this information is stored in the contract. Following which, they make another bid on line 31, and the contract must ensure that the bid is not withdrawn, and that the values do not become default struct values, as demonstrated on lines 33 – 34.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.7\_MultipleWithdraw\1.7.7\_NoDuplicateBidsAreAllowed.test.js*

**Test** **-** **1.7.8\_CantWithdrawABidThatsNotYours.test**  
 **🡺** **What it tests:**

* *Test that an investor may not withdraw another investor’s bid, even when knowing their hash*

**🡺** **Importance:**

* The contract must not allow other investors to cheat and manipulate the bids of an investor, as this would severely compromise the security of the contract’s bidding functionality.

**🡺** **How it works:**

* Two investors are involved in this test. The first investor places a valid bid on line 27. The test will ensure that their bid data is saved on lines 31 – 32. After this, the second investor will attempt to withdraw the bid, placed by the first investor, by using 1 as the \_remove value. It is expected that the transaction reverts because the contract will ensure that the address attached to the bid, stored in roundOneData, matches the address of the function caller. If not, then the revert happens.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.7\_MultipleWithdraw\1.7.8\_CantWithdrawABidThatsNotYours.test.js*

**Test** **-** **1.7.9\_CantWithdrawAfterRound1.test**  
 **🡺** **What it tests:**

* *Test that an investor cannot withdraw after the round 1 deadline*

**🡺** **Importance:**

* The contract must not allow investors to withdraw their bids after the round 1 deadline, as per NeverPay’s requirement, outlined on spec requirement .7.

**🡺** **How it works:**

* An investor places a valid bid during round 1, and the test ensures that the bid is stored, on lines 39 – 40. After this, forward time by the amount required to pass round 1 deadline, and the investor will attempt to withdraw the bid by using 1 as \_remove value. However, this is expected to revert, as demonstrated on line 44. It reverts, because round 1 deadline has passed, and withdraws are no longer allowed. If it does not revert, then the spec requirement is not satisfied.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.7\_MultipleWithdraw\1.7.9\_CantWithdrawAfterRound1.test.js*

**Test** **-** **1.8.1\_BidNotPlacedInRound1CantBeRevealed.test**  
 **🡺** **What it tests:**

* *Test that a bid that was not placed in round 1 cannot be revealed in round 2*

**🡺** **Importance:**

* It is important to ensure that a bid, whose order is not recorded, cannot be used to purchase shares in the second round, because this disallows the basic principle of checking for the bid order, during the share claiming phase (after round 2). Moreover, their blinded bid is never stored in the contract, which ignores the blinded aspect of the system.

**🡺** **How it works:**

* No investor places any bids in round 1, and after round 1 deadline has passed (it is round 2), an investor attempts to purchase shares, as shown on line 41, and this should revert the transaction, because the investor had not placed a bid to represent the shares they were intending to purchase, in round 1.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.1\_BidNotPlacedInRound1CantBeRevealed.test.js*

**Test** **-** **1.8.2\_WithdrawnBidCantBeRevealed.test**  
 **🡺** **What it tests:**

* *Test that a withdrawn bid cannot be revealed in round 2*

**🡺** **Importance:**

* If a bid was withdrawn, and an investor no longer has interest to purchase shares, then the investor should not be able to use this bid anymore, as per NeverPay requirement.

**🡺** **How it works:**

* An investor places and withdraws a bid in round 1. Thereafter, during round 2, they attempt to purchase shares for the bid they had withdrawn in round 1. As such, this should be reverted, as shown on line 45, because an investor may not purchase shares that are attached to a withdrawn bid (which is represented by an order = 0 in roundOneData).

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.2\_WithdrawnBidCantBeRevealed.test.js*

**Test** **-** **1.8.3\_InvestorBidDataIsRevealed.test**  
 **🡺** **What it tests:**

* *Test that the investor's bid data is revealed in round 2*

**🡺** **Importance:**

* As this is a blinded bid auction, during round 2, it is important that the information is revealed (in order to verify the bid hash) and to make this bid information public to other investors.

**🡺** **How it works:**

* During round 1, the investor places a bid. During round 2, they call the purchaseShares() function which requires them to provide, accurately, the information they had used to create the bid hash in round 1. This includes the number of shares they wish to purchase along with the ether they want to offer per share, and the nonce used to randomise the hash. If this information matches the information provided in round 1, then this data is stored in roundTwoData. As shown on lines 46 – 47, this information is clearly visible in this struct.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.3\_InvestorBidDataIsRevealed.test.js*

**Test** **-** **1.8.4\_InvestorCantCheatWhenRevealing.test**  
 **🡺** **What it tests:**

* *Test that the information that the investor reveals cannot be different from their initial bid*

**🡺** **Importance:**

* This is important due to the competitive nature of blinded bidding. If the investor finds that their bid is no longer competitive enough to keep up with the high prices or low order of other investors, they may not change their bid information to reflect a new price which may give them the advantage, as this would be cheating.

**🡺** **How it works:**

* An investor places valid bid in round 1 and during round 2, they generate another bid hash, and attempt to pass this information to the purchaseShares() function. Thus, this will result in a revert because the information does not correlate to any bid hash that exists in roundOneData.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.4\_InvestorCantCheatWhenRevealing.test.js*

**Test** **-** **1.8.5\_CantRevealAnotherInvestorBid.test**  
 **🡺** **What it tests:**

* *Test that an investor cannot reveal another investor's bid*

**🡺** **Importance:**

* It is a huge security breach if another investor was able to manipulate somebody else’s bid or reveal their hidden information before the investor intended to. Hence, investors should only be able to reveal their own bids.

**🡺** **How it works:**

* An investor places a bid in round 1, and during round 2, they attempt to reveal the bid of another investor. This is not possible, indicated by line 45, where the test expects the transaction to revert, as the contract ensures that the roundOneData.id == msg.sender for the caller of the purchaseShares() function.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.5\_CantRevealAnotherInvestorBid.test.js*

**Test** **-** **1.8.6\_BidhintMustExistInTheChain.test**  
 **🡺** **What it tests:**

* *Test that any bid, other than the first bid to be revealed, will require a valid bid hint parameter*

**🡺** **Importance:**

* In order to make the process of bidding fair, an investor, who is not the first person to purchase shares, will need to provide a valid bid hint (that exists in the system) so they may reduce their gas cost, and maintain a fair environment, payment wise.

**🡺** **How it works:**

* There are 2 investors in this iteration of the bidding process. They both place valid bids and during round 2, the first investor provides a bid hint which does not exist in the system. This is fine, because their bid is the first one in the entire linked list, so a bid hint is not expected to exist. However, the second investor will provide a random value, such as nonce1, which is not a valid bid hint that exists in the system, hence the transaction will be reverted, as seen on line 54.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.6\_BidhintMustExistInTheChain.test.js*

**Test** **-** **1.8.7\_CantRevealSameBidMultipleTimes.test**  
 **🡺** **What it tests:**

* *Test that the investor cannot reveal the same bid multiple times*

**🡺** **Importance:**

* By revealing the same bid multiple times, the investor also pays for the shares and is allocated this number of shares, which should not be allowed because the bid will always have the same order, meaning that, if that particular bid was successful, it would always be successful. Whereas, if it were a fair system, they would have to place multiple bids, causing the bids to have different order number, so one bid may be valid, while the other may not be, due to high competition.

**🡺** **How it works:**

* An investor places a valid bid in round 1, and during round 2, they try to purchase shares as shown on line 44. This should be successful, because the information they provide is correct. However, on line 45, they attempt to call the function again with the same bid hash. This time, the transaction will be reverted because the bid has already been paid for, and the investor is allocated shares for that particular bid.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.7\_CantRevealSameBidMultipleTimes.test.js*

**Test** **-** **1.8.8\_FirstBidIsHeadAndTailNode.test**  
 **🡺** **What it tests:**

* *Test that, if only one bid exists, it is equal to the tail and head node*

**🡺** **Importance:**

* Due to the linked list data structure, it is important that the first bid in the linked list is both the head and tail node, as no other bid exists in the list.

**🡺** **How it works:**

* An investor makes a valid bid during round 1 and during round 2, they purchase shares. Their bid is inserted into the linked list onchain and sorted appropriately. However, because this is the first bid in roundTwoData, there is no bid that is greater or lesser than in value, compared to it. Therefore, this bid is both the head and tail node. This is checked on line 47, which is expected to pass, where firstBid equals lastBid.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.8\_FirstBidIsHeadAndTailNode.test.js*

**Test** **-** **1.8.9\_OrderABidLowerThanTopBid.test**  
 **🡺** **What it tests:**

* *Test that a bid lower than the top bid (head node) will be ordered lower than it*

**🡺** **Importance:**

* This is an important test to ensure that the linked list sorting mechanism works appropriately, as a bid with less value than the head node (lowest valued bid) will eventually resume a spot under the head node, because this new bid is less valued. Thus, ensuring that the list is always sorted.

**🡺** **How it works:**

* There are 2 investors who place bids during round 1. During round 2, they attempt to purchase shares, and, while the 1st investor provides a random bid hint, the 2nd investor will provide the 1st investor’s bid hash as the bid hint, because it’s the only bid that exists in the linked list. Hence, the 2nd investor’s bid will be ordered as lower than the 1st investor. This is because, when the investors placed bids in round 1, first investor paid 1 ether per share, and second investor paid 2 ether per share. However, in the for loop starting on line 45, the order that investors purchase shares is reversed, meaning that the second investor purchases shares first, making them firstBid and lastBid. The first investor then purchases shares, and because they only pay 1 ether per share, they can be behind the 2nd investor, in the list. This test is checked on line 61, to ensure that the 1st investor is prior to the 2nd investor.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.9\_OrderABidLowerThanTopBid.test.js*

**Test** **-** **1.8.10\_OrderABidHigherThanTopBid.test**  
 **🡺** **What it tests:**

* *Test that a bid higher than the top bid (tail node) will be ordered higher than it*

**🡺** **Importance:**

* This test ensures that the linked list works appropriately, because a bid which has higher value than the current highest valued bid in the list should resume a position above the previously highest valued bid, ensuring that this new bid will be the new tail node

**🡺** **How it works:**

* 2 investors place bids in round 1, where the investors who place bids later on pay more ether per share, and thus, have higher valued bids. During round 2, both investors purchase shares in the order that they placed bids, and as such, the 2nd investor’s bid has a higher value than the 1st investor’s bid. Hence, 1st bidder’s bid should have a bid.next that points to the higher valued bid, which is 2nd investor’s bid. This is tested on line 61.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.10\_OrderABidHigherThanTopBid.test.js*

**Test** **-** **1.8.11\_OrderABidInTheMiddle.test**  
 **🡺** **What it tests:**

* *Test that a bid that belongs in the middle of the list, is ordered appropriately*

**🡺** **Importance:**

* If a bid is to be sorted in the middle of the list, it is important that this bid is able to find a position where its value is less than the bid ahead of it (bid.next), but also higher than the value of the bid prior to it (bid.prior).

**🡺** **How it works:**

* There are 4 investors who place bids, and purchase shares with value of each investor’s bid increasing. A 5th investor also places a bid and decides to purchase shares after all other investors have purchased shares. The 5th investor’s bid value is of higher value compared to investors 1 and 2. But is lower valued than investors 3 and 4. Hence, this 5th investor’s bid must sit between the values. Meaning, that the 5th investor’s bid will have both a bid of higher value and a bid of lower value attached to it in the linked list. This is test on lines 97 and 98, to ensure that the 5th investor’s bid is not head or tail node, but in fact, a node in the middle.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.11\_OrderABidInTheMiddle.test.js*

**Test** **-** **1.8.12\_OrderABidLowerThanLowestBid.test**  
 **🡺** **What it tests:**

* *Test that a lower bid, than current head node, will become the new lowest bid (head node)*

**🡺** **Importance:**

* Unlike test 1.8.9, this test ensures that the new head node (which is the current bid) and also the lowest valued bid, will definitely resume the spot of the head node, making it the new head.

**🡺** **How it works:**

* There are 4 investors who place bids and purchase shares. There is also a 5th investor who places a bid and purchases shares. However, their bid value is lower than every other investor. As such, it is tested on lines 100 and 101 that their bid becomes the head node, by ensuring that the bid.next has a link (meaning there are bids of higher value), while the bid.prior has no link (is equal to 0), meaning there are no bids which have a lower value.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.12\_OrderABidLowerThanLowestBid.test.js*

**Test** **-** **1.8.13\_CanHoldOver100Bids.test**  
 **🡺** **What it tests:**

* *Test that over 100 bids may be stored in the list, to show scalability*

**🡺** **Importance:**

* This is somewhat of a scalability test, which ensures that regardless of the number of bids in the system, that all investors will pay relatively equally. It is important to ensure that this test is conducted without a timeout, as it will take time to conduct over 200 transactions.

**🡺** **How it works:**

* By using a sample of 101 investors, the test ensures that all investors may place bids, and that all investors may purchase shares successfully, without encountering a revert, due to an error with the linked list, or other functional logic in the contract.

**Requirements:** *default requirements ALSO: (increase accounts to 110 on ganache-cli (by using -a 110))*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.13\_CanHoldOver100Bids.test.js*

**Test** **-** **1.8.14\_NumberOfBidsIsStoredAccurately.test**  
 **🡺** **What it tests:**

* *Test that the variable that represents bids size is represented accurately by total number of bids*

**🡺** **Importance:**

* As this variable, that represents the total paid bids in the system, is used throughout the contract, it is essential that this variable accurately depicts the number of bids in the list.

**🡺** **How it works:**

* Using a sample of 15 investors, who place bids and purchase shares, the test ensures that the value of paidBidCount (in the contract) is equivalent to the number of bids that exist in the linked list. This is apparent on line 84, where the numOfBids (which is the value of paidBidCount) is compared to the total investors (15). It is expected that both are equal, as the placing and purchasing logic is iterated 15 times through loops.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.14\_NumberOfBidsIsStoredAccurately.test.js*

**Test** **-** **1.8.15\_HeadAndTailNodeDifferAfterMultipleBids.test**  
 **🡺** **What it tests:**

* *Test that the head and tail nodes are not the same after more than 1 bid exists*

**🡺** **Importance:**

* After more than 1 bid exists, it’s important to understand that there will be a bid of greater value and a bid of lower value. As such, the tail node cannot be the head node.

**🡺** **How it works:**

* Using a sample data of 15 investors, after multiple bids, it is certain that the head and tail node cannot be the same, because there exist bids which are of greater and lower value within the linked list. This is tested by lines 82 – 84, to ensure that the firstBid and lastBid are not the same after 15 investors have placed bids and purchased shares.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.15\_HeadAndTailNodeDifferAfterMultipleBids.test.js*

**Test** **-** **1.8.16\_NothingLowerThanHeadNode.test**  
 **🡺** **What it tests:**

* *Test that the linked list works as intended, by ensuring that there is no lower bid than head node*

**🡺** **Importance:**

* Due to the nature of a linked list, nothing can exist prior to the head node, as this is the lowest node. In this environment, it would be the bid of the lowest value, which cannot have a bid of lower value as its bid.prior.

**🡺** **How it works:**

* After investors have attempted to purchase shares, where the very first investor is the lowest valued bidder, and the last investor is the highest valued bidder, the test ensures that the investor who paid the least ether per share remains as the lowest valued bid (head node) on lines 83 – 85. Moreover, it ensures that there is no other value lower than the current head node.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.16\_NothingLowerThanHeadNode.test.js*

**Test** **-** **1.8.17\_NothingHigherThanTailNode.test**  
 **🡺** **What it tests:**

* *Test that the linked list works as intended, by ensuring that there is no higher bid than tail node*

**🡺** **Importance:**

* Similarly, a tail node represents the highest valued bid that an investor has paid for. Nothing may exist above it (bid.next) as there is no other bid which is of higher value. This test must ensure that this is the case, and thus, will ensure that the linked list works appropriately.

**🡺** **How it works:**

* After investors have purchased shares, this test ensures that the 15th investor (last investor to purchase shares and also the highest paying investor) will inevitably be the tail node of the linked list, by ensuring that there is no other bid that exists above lastBid (bid.next = 0). This is tested on lines 83 – 85.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.8\_Round2\1.8.17\_NothingHigherThanTailNode.test.js*

**Test** **-** **1.9.1\_EtherEqualToSharesTimesPrice.test**  
 **🡺** **What it tests:**

* *Test that the investor pays enough ether to cover costs for the shares they are buying*

**🡺** **Importance:**

* It is pivotal that NeverPay receives the money that is owed to them by the offer that the investor makes. As such, the investor is required to pay shares \* price worth of ether, as per their bid.

**🡺** **How it works:**

* Investor places a bid and purchases shares in round 2. During this purchasing process, the contract must ensure that the value of the deposit of Ether that the investor makes is price \* shares \* 1 ether, which is 10^18 wei. This is tested on line 44.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.9\_Payment\1.9.1\_EtherEqualToSharesTimesPrice.test.js*

**Test** **-** **1.9.2\_InvestorEtherIsStoredInTheContract.test**  
 **🡺** **What it tests:**

* *Test that the investor's deposit is stored in the smart contract*

**🡺** **Importance:**

* Is it absolutely important that the investor’s ether is stored on the contract because if they overpay or are unsuccessful, NeverPay must refund the investor the investor’s ether.

**🡺** **How it works:**

* An investor places a bid and attempts to purchase shares. During this process, the test ensures that the contract stores the value of the deposit that the investor makes, in the roundTwoData.deposit variable. This is tested on line 47.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.9\_Payment\1.9.2\_InvestorEtherIsStoredInTheContract.test.js*

**Test** **-** **1.9.3\_NotEnoughEtherMeansItsNotTaken.test**  
 **🡺** **What it tests:**

* *Test that if not enough Ether is provided, that their Ether is not taken*

**🡺** **Importance:**

* NeverPay must ensure that they receive the money owed to them for the value of their shares, which is determined by the investor’s bid, and the price they’re willing to pay per share. However, if the investor does not give enough ether to cover this cost, then the bid may not be accepted, and the investor may not purchase shares until they have enough ether.

**🡺** **How it works:**

* After an investor places a bid and attempts to purchase shares, it is expected that the transaction reverts because the value of the ether they are paying to the contract is reduced by 2000, compared to the value of shares \* price \* 1 ether. This is shown on line 44, and it means that the investor has not provided enough ether toc over the cost of the shares they want to purchase. Hence, revert.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.9\_Payment\1.9.3\_NotEnoughEtherMeansItsNotTaken.test.js*

**Test** **-** **1.9.4\_EtherOverpayIsFullyAccepted.test**  
 **🡺** **What it tests:**

* *Test that an overpay is accepted completely*

**🡺** **Importance:**

* If an investor pays more ether than their shares are worth, then NeverPay may not accept this money, and must return it back to the investor. However, this may be done during the share claiming process, meaning that the contract can fully accept the overpay.

**🡺** **How it works:**

* All ether that is sent after an investor places a bid and attempts to purchase shares, is accepted. The test ensures that despite the investor paying more ether than they need to, that they are still able to purchase shares. This is evident on line 44.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.9\_Payment\1.9.4\_EtherOverpayIsFullyAccepted.test.js*

**Test** **-** **1.10.1\_*CantClaimSharesNotRevealedInRound1*.test**  
 **🡺** **What it tests:**

* *Test that investor can't claim shares that are not revealed in round 1*

**🡺** **Importance:**

* It should not be possible for investors to fabricate a bid that does not exist in the system, as this would ignore the rules of deadlines and the overall competitive environment that NeverPay intended.

**🡺** **How it works:**

* A bid is not made by any investor, however, after round 2 deadline, an investor attempts to claim shares for an unknown bid, as shown on line 38, but this transaction is reverted because the bid does not exist, and had not been placed in round 1.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.1\_CantClaimSharesNotRevealedInRound1.test.js*

**Test** **-** **1.10.2\_*CantClaimSharesNotRevealedInRound2*.test**  
 **🡺** **What it tests:**

* *Test that the investor can't claim shares that are not revealed in round 2*

**🡺** **Importance:**

* Similarly, if a bid is not revealed, and verified, it will ignore the blinded auction + revealing rule, and the overall competition that NeverPay intended. Hence this cannot be allowed.

**🡺** **How it works:**

* An investor makes a bid in round 1 and does not reveal it in round 2. After round 2 deadline has passed, the investor attempts to claim shares for a bid which has not been revealed, as shown on line 40. This is not accepted, and the transaction should revert, as the investor has clearly ignored the rules for round 2 (revealing their bid). Moreover, the investor has not paid any ether for the shares they wish to purchase.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.2\_CantClaimSharesNotRevealedInRound2.test.js*

**Test** **-** **1.10.3\_*InvestorCannotClaimSharesOfAnotherInvestor*.test**  
 **🡺** **What it tests:**

* *Test that the investor cannot claim another investor's shares*

**🡺** **Importance:**

* It is a huge security breach to allow another investor to gain the shares that another investor paid for. Hence this must not be allowed whatsoever.

**🡺** **How it works:**

* An investor places a bid, and purchases shares throughout the rounds. After round 2 deadline, another investor attempts to claim the shares for the first investor, by using their bid Hash. However, as shown on line 49, this transaction should be reverted, as the bid’s id (stored in roundOneData) does not match the address of the function caller, which is the other investor.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.3\_InvestorCannotClaimSharesOfAnotherInvestor.test.js*

**Test** **-** **1.10.4\_*CantClaimSharesOrRefundTwice*.test**  
 **🡺** **What it tests:**

* *Test that the investor can't claim shares multiple times*

**🡺** **Importance:**

* Investors should not be able to refund or claim shares multiple times for a single bid, because then they would be trying to cheat the system and/or trying to empty the contract’s ether balance (stealing).

**🡺** **How it works:**

* An investor places a bid and purchases shares. After round 2 deadline, they claim their shares for the first time, and this is successful, as they are a successful and competitive bidder. However, when they attempt to claim their shares again, the transaction is reverted because they have already claimed their shares, allowing their deposit = 0, and the contract checks that deposit not = 0. The transaction revert test is evident on line 50.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.4\_CantClaimSharesOrRefundTwice.test.js*

**Test** **-** **1.10.5\_*OverpayingInvestorGetsRefundThatIsOwed*.test**  
 **🡺** **What it tests:**

* *Test that the overpaying investor gets a full refund of what they're owed (extra payment)*

**🡺** **Importance:**

* If an investor, who is successful, pays extra ether for shares, compared to the value of the shares they purchase, then this money should be returned to the investor, as they are not using the ether to purchase more shares.

**🡺** **How it works:**

* An investor places a bid and purchases shares, while depositing more ether than the value of the shares they are purchasing. Hence, when they claim the shares, after they receive the shares, they are also refunded the amount of money they are owed, by the contract. This is evident on line 50, which ensures that their balance is an overpaid amount. And then at line 54, the test checks that their new balance after claiming shares, is equal to their overpay – total cost, and also that the balance is not 0, meaning that they were refunded money.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.5\_OverpayingInvestorGetsRefundThatIsOwed.test.js*

**Test** **-** **1.10.6\_*FirstInvestorWillFindLowestValidBid*.test**  
 **🡺** **What it tests:**

* *Test that first investor claiming shares will find lowest valid bid*

**🡺** **Importance:**

* It is the duty of the first investor, who is claiming shares, to find the lowest valid bid in the linked list, as this will allow all other investors to have an equal cost for claiming shares.

**🡺** **How it works:**

* Investors will place bids and purchase shares. After round 2 deadline, an investor will claim shares, and this first investor will find the lowestValidBid, which is initially 0. However, after they claim their shares, this value is no longer 0, meaning it has been found. This is evident on line 50.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.6\_FirstInvestorWillFindLowestValidBid.test.js*

**Test** **-** **1.10.7\_*LowestValidBidFinderCanBeIdentified*.test**  
 **🡺** **What it tests:**

* *Test that the bidder with the lowest valid bid can be identified to receive incentives*

**🡺** **Importance:**

* As the first investor to claim shares is revealing the lowest valid bidder, and is also responsible for doing more computation, increasing their gas cost for the transaction, it is important that they can be tracked by NeverPay, so that the company can give them an incentive / reward for computing this cost.

**🡺** **How it works:**

* After bids are placed and shares are purchased, during the claiming process, the first investor will find the lowest valid bidder. However, to ensure that this investor is tracked, the lowestValidBidHashFinder value determines the bid hash of this investor. The test will ensure that it is not a default value 0, on line 51, meaning that this investor’s hash is, in fact, saved and can now be tracked by NeverPay.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.7\_LowestValidBidFinderCanBeIdentified.test.js*

**Test** **-** **1.10.8\_*ValidBidHigherThanLowestCanClaimShares*.test**  
 **🡺** **What it tests:**

* *Test that a valid bidder, higher than lowest valid bid, can claim shares*

**🡺** **Importance:**

* It is important that anybody who has placed a more competitive bid than the lowest valid bid is eligible to receive shares.

**🡺** **How it works:**

* After investors have placed bids and purchased shares, the test checks whether an investor who is defined to be competitive enough is able to receive shares by calling the claimShares() function on line 88, and then checking if their balance of shares is not 0, meaning that they received their shares, and meaning that they are successful. This is done on line 91.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.8\_ValidBidHigherThanLowestCanClaimShares.test.js*

**Test** **-** **1.10.9\_*LowestValidBidderCanClaimShares*.test**  
 **🡺** **What it tests:**

* *Test that the lowest valid bidder can claim shares*

**🡺** **Importance:**

* It is important to ensure that the lowest valid bidder is, as the name implies, the lowest VALID bidder, so they are still able to claim their shares. The test must ensure this.

**🡺** **How it works:**

* After investors have placed bids and purchased shares, the lowest valid bidder, being investor number 4 or accounts[5], will attempt to claim their shares on line 88. Then, on line 91, the test ensures that their balance is no longer 0, meaning that they received shares from the claiming process, thus implying that they are successful.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.9\_LowestValidBidderCanClaimShares.test.js*

**Test** **-** **1.10.10\_*LowerThanLowestValidCantClaimShares*.test**  
 **🡺** **What it tests:**

* *Test that the bidder lower than lowest valid bid cannot claim shares*

**🡺** **Importance:**

* Anyone that is not competitive enough is unable to receive shares. This is an important competitive factor set by NeverPay’s auction requirement and this test will ensure that it works.

**🡺** **How it works:**

* Investors will purchase shares, and during the claiming process, a bid that is defined to be less valuable than the lowest valid bid will try to claim shares on line 88. However, as shown on line 91, they are expected to be unable to receive any shares, hence the test checks that assert.equal(0), meaning they receive no shares.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.10\_LowerThanLowestValidCantClaimShares.test.js*

**Test** **-** **1.10.11\_*NotAllSoldAllBiddersGetWhatTheyWanted*.test**  
 **🡺** **What it tests:**

* *Test that all bidders get the shares they wanted to purchase, when not all shares sold*

**🡺** **Importance:**

* It is important to ensure that if not all shares are sold, meaning the supply is still greater than demand, then all the investors who have purchased shares get what they wanted.

**🡺** **How it works:**

* After investors purchase shares, during the claiming phase, using a loop, the test checks that after the investor has claimed their shares by calling the function, that their balance is incremented by the shares they intended to purchase, which they reveal in round 2. The assertion takes place at line 94.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.11\_NotAllSoldAllBiddersGetWhatTheyWanted.test.js*

**Test** **-** **1.10.12\_*AllSharesSoldLowestValidBidderGetsLeftovers*.test**  
 **🡺** **What it tests:**

* *Test that when all shares are sold, the lowest valid bidder gets leftovers*

**🡺** **Importance:**

* The lowest valid bidder is still a valid bidder, meaning that they are competitive enough to receive shares. However, due to not being as competitive as the bids above the investor’s bid, it is intended that NeverPay will give this investor the remaining bids, which will be left over, if all other investors claimed their bids.

**🡺** **How it works:**

* The investors purchase shares, and during the claiming process, between lines 94 – 99, the test ensures that those who are highly competitive get all the shares they wanted, while those who are not competitive enough, compared to the demand, do not receive shares. Moreover, the 4th investor, who is the lowest valid investor, on line 96 – 97, will receive what ever is left over, which is 1000 shares, even though this investor wanted 1500 shares, due to the lower competitiveness of their bid.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.12\_AllSharesSoldLowestValidBidderGetsLeftovers.test.js*

**Test** **-** **1.10.13\_*LowestValidBidderGetsEnoughSharesAllSold*.test**  
 **🡺** **What it tests:**

* *Test that the lowest valid bidder gets all shares they bought if enough supply is available*

**🡺** **Importance:**

* If there is enough supply, even though all the shares are sold, the lowest valid bidder should still receive all the shares they wanted to purchase.

**🡺** **How it works:**

* After investors purchase shares, and during the claiming process, ensure that all investors receive 1000 shares, as shown on line 93, because there are 10 investors, who all wanted 1000 shares, so 10 \* 1000 = 10,000, which meets the maximum supply, instead of exceeding it.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.10\_RoundsEnded\1.10.13\_**LowestValidBidderGetsEnoughSharesAllSold.test.js*

**Test** **-** **1.11.1\_*AllInvestorsHaveUniqueOrderNumber*.test**  
 **🡺** **What it tests:**

* *Test that all bids are labelled with a different (ascending) order number*

**🡺** **Importance:**

* It is very important that no order number is the same, as no bid can be placed simultaneously at the same time, and it is intended that no bid is ever equal to another bid.

**🡺** **How it works:**

* After investors place bids, the test collects the order number of all the bids, and every time a new bid is placed, its order number is checked to ensure that no other bid has the same order number, as shown on line 43 – 44. If this is the case, then every order number is unique.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.11\_BidTies\1.11.1\_AllInvestorsHaveUniqueOrderNumber.test.js*

**Test** **-** **1.11.2\_*SortingByOrderIsMaintained*.test**  
 **🡺** **What it tests:**

* *Test that order number is always increasing, as more bids are placed*

**🡺** **Importance:**

* Order number cannot decrease because the order will always increase as new bids are placed or withdrawn. If order number ever decreased, then it would create unfairness, as investors who bid later are more competitive than investors who bid earlier, which doesn’t make sense.

**🡺** **How it works:**

* Every time investors place bids, their order number is stored, and the next bid is checked against the previous order number to see if bid\_2 order number = bid\_1 order number + 1, meaning that the order is maintained in an ascending manner.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.11\_BidTies\1.11.2\_SortingByOrderIsMaintained.test.js*

**Test** **-** **1.12.1\_*UnsuccessfulBiddersGetRefunds*.test**  
 **🡺** **What it tests:**

* *Test that unsuccessful bidders will receive refunds*

**🡺** **Importance:**

* Investors who are not competitive enough but send money to the contract are able to get a refund for the value of the shares that were not able to successfully acquire. If they over pay, then all of the overpay amount is returned.

**🡺** **How it works:**

* After investors have purchased shares, during the claiming process, the unsuccessful investors, which are the bottom 3 investors in this case, shown at line 103, will be given a refund. After they call claimShares on line 100, their balance is checked in line 105 to ensure that the balance they have after calling claimShares is greater than the balance they had after sending ether. This means that the contract was able to refund them for the ether that they paid to the contract. This assert is made on line 108.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.12\_Unsuccessful\1.12.1\_UnsuccessfulBiddersGetRefunds.test.js*

**Test** **-** **1.13.1\_TotalSupplyIsSumOfSharesClaimed.test**  
 **🡺** **What it tests:**

* *Test that the total share supply is equal to the amount of shares that are successfully sold and claimed*

**🡺** **Importance:**

* To comply with ERC-20 standards, it’s required that the total supply of the token (shares) is equivalent to the number of shares that are available in circulation, to be transferred, and this would be the number of shares that have been claimed.

**🡺** **How it works:**

* After investors have placed bids, and purchased shares, during the share claiming process, the contract adds the total number of shares that the investor successfully purchases to the investor’s balance, and to the totalSupply. This can be checked at line 97 – 98, where the test ensures that the totalSupply is equivalent to all the shares sold to investors in this test.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.13\_Unsuccessful\1.13.1\_TotalSupplyIsSumOfSharesClaimed.test.js*

**Test** **-** **1.13.2\_InvestorCanApproveAllowance.test**  
 **🡺** **What it tests:**

* *Test that an investor can approve some allowance of shares for another investor*

**🡺** **Importance:**

* Complying with the ERC-20 standard, investors are eligible to approve an allowance for another investor to spend on their behalf. In this case, this would be the shares that other addresses may spend on the investor’s behalf.

**🡺** **How it works:**

* After shares are purchased, and investors claim shares, it can be seen at line 93, that investor 1 approved 250 shares for investor 2. To ensure that this functionality works, from lines 95 – 96, the test ensures that investor 2 has successfully been given the allowance of 250 shares.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.13\_Unsuccessful\1.13.2\_InvestorCanApproveAllowance.test.js*

**Test** **-** **1.13.3\_InvestorCanTransferSharesToAnother.test**  
 **🡺** **What it tests:**

* *Test that the investor can transfer their shares directly over to another investor*

**🡺** **Importance:**

* It is crucial function, as it complies with ERC-20 standards, and NeverPay has explicitly required that investors are able to transfer their shares to others after the round 2 deadline.

**🡺** **How it works:**

* After shares are purchased and claimed, the testing begins from line 93 – 94 which stores that previous shares of both investors, then after investor 1 transfers 300 shares to investor 2, from line 98 – 99 is the new amount of shares that each of the investor has. If transfer were to be successful, the test would expect that investor 1 now has 300 less shares, and investor 2 has 300 more shares than they both did before the transfer. This is tested on lines 101 – 102, and if the transfer is successful, this assertion would not throw an error.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.13\_Unsuccessful\1.13.3\_InvestorCanTransferSharesToAnother.test.js*

**Test** **-** **1.13.4\_InvestorBalanceIsIncreasedWhenTheyClaimShares.test**  
 **🡺** **What it tests:**

* *Test that the investor's shares balance is increased by the number of shares they successfully claim*

**🡺** **Importance:**

* It is crucial to increase the investor’s balance by the shares they purchase, as this is how they will be able to give allowance/transfer shares after round 2 ends, and the ERC-20 standards come into effect.

**🡺** **How it works:**

* After shares are purchased and claimed, it is tested that each investor’s balance is incremented by the equivalent number of shares that they intended to buy in the reveal round. This is tested on the assertion on line 94.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.13\_Unsuccessful\1.13.4\_InvestorBalanceIsIncreasedWhenTheyClaimShares.test.js*

**Test** **-** **1.13.5\_InvestorCanTransferAfterApprovalAndAllowance.test**  
 **🡺** **What it tests:**

* *Test that an investor may successfully transfer shares that they are approved by another investor*

**🡺** **Importance:**

* This is a crucial ERC-20 standard which must be met, where an address which is given approval to spend an allowance is able to do so.

**🡺** **How it works:**

* Once investors purchase shares and claim them, an investor gives another investor an allowance of 250 shares, as shown on line 93. Line 96 ensures that this investor has received an allowance of 250 shares. Afterwards, the investor transfers 150 of the 250 shares to another address, and now the test will ensure that 150 shares have been removed from the investor who gave the allowance, leaving them with 350 shares (initially 500 shares). Moreover, the test ensures from line 102 – 103, that the other address had received these 150 shares, and now have 650 shares (initially 500 shares). After which, it is tested that the address which has allowance now has 250 – 150 = 100 remaining shares to be able to transfer on behalf of the investor who gave the allowance, as shown on line 106 (this is tested).

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.13\_Unsuccessful\1.13.5\_InvestorCanTransferAfterApprovalAndAllowance.test.js*

**Test** **-** **1.13.6\_InvestorCantTransferMoreThanTheyHave.test**  
 **🡺** **What it tests:**

* *Test that the investor can't transfer more shares than they currently own*

**🡺** **Importance:**

* It should not be possible for any investor to give what they don’t have – meaning that they cannot give 100 shares to an investor when they have 0 shares or less than 100 shares. Otherwise, this is a form of ‘cheating’ the system and gaining free shares.

**🡺** **How it works:**

* Once shares have been bought and claimed, the test ensures that if an investor has bought 500 shares, and have a balance of 500 shares, then they are unable to transfer more than 500 shares to anyone. This is tested on line 94, where the test expects transaction to be reverted. The reason that this doesn’t work is that if they wanted to transfer 1000 shares, it would be 500 – 1000 = -500, and because the value of an investor’s balance is stored as a uint, it cannot be a negative number. Therefore, it reverts.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\1\_NeverPayShares\1.13\_Unsuccessful\1.13.6\_InvestorCantTransferMoreThanTheyHave.test.js*

**Test** **–** **2.1.1\_NonexistantKeysAreNotAuthorized.test**  
 **🡺** **What it tests:**

* *Test that keys that don't exist at all, are not authorized*

**🡺** **Importance:**

* It should not be possible for investors to authorize themselves by testing against a key which does not exist in ASIC’s registry, as this would be forging / cheating the system.

**🡺** **How it works:**

* On line 14, the signerAuthorityAccount.address is validated as a key in the registry, however, in line 15, a different key accounts[3] is used, which clearly does not exist in the registry. Hence, it is expected that on line 16, keyExists is false, because the key accounts[3] does not exist in the registry.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.1\_Registry\2.1.1\_NonExistantKeysAreNotAuthorized.test.js*

**Test** **–** **2.1.2\_ExistingKeysAreAuthorizedInTheRegistry.test**  
 **🡺** **What it tests:**

* *Test that existing keys are authorized in the registry*

**🡺** **Importance:**

* It is also important to ensure that if an investor provides a key that is authorized, then they are able to prove that they are a sophisticated investor.

**🡺** **How it works:**

* The signerAuthorityAccount.address key is added to the registry, and on line 15, the test checks if this key exists. Therefore, it clearly does exist as it has been added to the registry via addKeyToRegistry(). Hence, the assertion on line 16 should be true, meaning that the key exists.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.1\_Registry\2.1.2\_ExistingKeysAreAuthorizedInTheRegistry.test.js*

**Test** **–** **2.1.3\_ASICCanAddMoreKeysToRegistry.test**  
 **🡺** **What it tests:**

* *Test that ASIC can add more keys to the registry*

**🡺** **Importance:**

* ASIC can add as many keys as it wants to the registry, and it created and manages. This is the fundamental requirement for the ASIC contract.

**🡺** **How it works:**

* ASIC simply adds keys through the addKeyToRegistry() function, and since the function caller address is ASIC itself, which is a requirement, then the key will be added, along with the financial organization’s name, to a mapping.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.1\_Registry\2.1.3\_ASICCanAddMoreKeysToRegistry.test.js*

**Test** **–** **2.1.4\_*ASICCanRemoveExistingKeysFromRegistry*.test**  
 **🡺** **What it tests:**

* *Test that ASIC can remove existing keys from the registry*

**🡺** **Importance:**

* ASIC also can remove keys from the registry that it created. This is another fundamental requirement for the ASIC contract.

**🡺** **How it works:**

* ASIC adds multiple organization keys by calling the contract as ASIC, and then is able to remove these keys by calling the removeKeyFromRegistry() function, which allows it to remove keys that exist in the registry, if found. The function caller must, once again, be ASIC as they are the only approved address to access the mapping / registry.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.1\_Registry\2.1.4\_ASICCanRemoveExistingKeysFromRegistry.test.js*

**Test** **–** **2.1.5\_RandomPersonCantAddKeyToRegistry.test**  
 **🡺** **What it tests:**

* *Test that a random person can't add keys to the registry*

**🡺** **Importance:**

* It would be a huge security breach of other addresses were able to add keys, as investors could simply add a random key to the registry in order to authenticate themselves. Hence this should not be allowed.

**🡺** **How it works:**

* After ASIC adds the CommBank key to the registry, on line 16, a random person calls the addKeyToRegistry() function using their own address, and try to insert another organization key into the registry. This, however, is not possible because the contract checks that the function caller is ASIC, in order to manage the registry. Hence, the expected result is that the transaction will revert, as tested on line 17.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.1\_Registry\2.1.5\_RandomPersonCantAddKeyToRegistry.test.js*

**Test** **–** **2.1.6\_RandomPersonCantRemoveKeyFromRegistry.test**  
 **🡺** **What it tests:**

* *Test that a random person can't remove keys from the registry*

**🡺** **Importance:**

* It would be another huge breach if a random person were able to simply remove keys from ASIC’s registry. Hence, this is also not allowed

**🡺** **How it works:**

* After ASIC adds Commbank key to the registry, on line 16, a random person tries to remove a key by calling removeKeyFromRegistry(). However, this will be met with a transaction revert because the function was called with an address that is not ASIC’s (transaction not signed by ASIC). Hence, revert is expected, and security is maintained.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.2\_Registry\2.1.6\_RandomPersonCantRemoveKeyFromRegistry.test.js*

**Test** **–** **2.2.1\_ValidateInvestorSignatureProof.test**  
 **🡺** **What it tests:**

* *Test that NeverPay validates the investor's signature*

**🡺** **Importance:**

* NeverPay must ensure that its investors are sophisticated investors in order to abide by ASIC’s rules against illegal collection of money (as specified on the spec).

**🡺** **How it works:**

* The investor is issued a certificate, as shown on line 17, where they provide their address. After which, the signer authority’s key is added to the registry of authorized signers, as shown on line 22. Therefore, on line 23, the investor should be able to validate their status by providing the certificate that the authority issued.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.2\_Verification\2.2.1\_ValidateInvestorSignatureProof.test.js*

**Test** **-** **2.2.2\_CannotValidateUsingAnotherInvestorSignature.test**  
 **🡺** **What it tests:**

* *Test that an investor cannot validate himself using another investor's signature*

**🡺** **Importance:**

* Investors, who may not be sophisticated investors, should not be able to authenticate their status using other investors’ certificates.

**🡺** **How it works:**

* An investor gains a certificate from an authority, as shown on line 19. However, on line 25, another investor attempts to use this certificate. It is expected that the transaction should be reverted, because the contract produces its own message using the function caller’s address, and creates an eth signed message internally. After which, it uses this message and the signature provided by the signer authority, to ensure that the address on the certificate matches the address of the function caller. This is not the case, hence, revert transaction.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.2\_Verification\2.2.2\_CannotValidateUsingAnotherInvestorSignature.test.js*

**Test** **–** **2.2.3\_CanUseCertificatesFromMultipleOrganizations.test**  
 **🡺** **What it tests:**

* *Test that an investor can use certificates issued from multiple authorities*

**🡺** **Importance:**

* It is important to allow investors to be able to prove their sophisticated investor status by using multiple organizations, because they may be financially inclined with multiple banks in Australia.

**🡺** **How it works:**

* Multiple organizations are created, and an investor gains a certificate from these organizations. During the bidding process, the investor is able to use any certificate he gains from these organizations (which are ASIC approved), to prove the investor’s sophisticated investor status, as shown on line 27.

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.2\_Verification\2.2.3\_CanUseCertificatesFromMultipleOrganizations.test.js*

**Test** **-** **2.2.4\_CantValidateUsingInvalidSignature.test**  
 **🡺** **What it tests:**

* *Test that an invalid signature, (that is not signed by an authority) can't be validated*

**🡺** **Importance:**

* An investor should not be able to validate their sophisticated investor status without using a valid signature that is issued by ASIC supported financial organizations.

**🡺** **How it works:**

* When an investor decides to make a bid, they supply a signature which does not contain a signed message and does not correlate to a signer’s address which exists in the registry This is shown by the revert at line 24, which is expected, because the investor may not validate themselves

**Requirements:** *default requirements*  
**Terminal:** truffle test *.\test\2\_SICAR\2.2\_Verification\2.2.4\_CantValidateUsingInvalidSignature.test.js*