# Hydrogen Platform
# DA Solidity Audit

***Samuel JJ Gosling***

September 11, 2018

## Overview

- Time Committed: 30 Hours
- A smart contract for the Hydrogen platform where community members delegate others to be a HYDRO ambassador, by proposing a subject address to act as an nominee. The actual vote is operated through the voting contract and not within the DA, it is to note that there is a reward/payout system for ambassadors and is definitely the most likely aspect to fall flaw to a security breach.

## Methodology

Approaches taken;

- Manual line by line analysis
- Solidity analysis through Solium
- Solidity analysis through Mythril OSS
- Symbolic functionality analysis via Manticore
- Manual functionality through creating a testing dashboard in React.JS

## Findings

- Line 89 - *library addressSet*
  - All function inputs are the *_addressSet* storage *struct* and an *address* for submission
    - Line 90 - *_addressSet struct*
      - **Speculation;** A solidified storage *struct* is defined here, which contains an array of addresses and a key mapping structure in order to keep track of the *uint* indexes of an address that is contained within the array for deleting elements accurately and without spending gas on iteration.
    - Line 116 - *function contains*()
      - **Speculation;** Contains an operation to verify the submission of the inputted address in the inputted *_addressSet* storage structure, if the index is greater than zero then it's apparent that it consists within the struct if so the return is *true* it not outputs *false*.
      - **Functionality;** Validates if the subject address is already an active element of the specified *_addressSet struct* by outputting a *bool* value to indicate it's authenticity based on the condition if the *members array.length* is greater than zero. Short address results in no notable discrepancies observed.
    - Line 120- *function length*()
      - **Speculation;** Is a mechanism to view the *uint* length of the address array contained within the inputted *_addressSet struct* to specify the new index of a new submission to the array.
      - **Functionality;** Executes as expected by returning the *_addressSet* struct address *array.length.* Short address emits no dysfunctional behaviour

- Line 95 - *function insert*()
  - **Speculation;** A function to insert a new submission to the array, it is handled with a condition that consists of on the basis of a *false* return via the *contain*() function to verify if the address is already an active element. If the result of execution is *true* the function terminates with no operation. There then is an *assert*() function condition to validate that there is enough space still within the struct in order to avoid overflow as the max length of an array is $2^{256}$ with only $2^{256} - 1$ indexes available for submission. The inputted array is then pushed to the address array within the struct and then following that the index is logged using the key mapping.
  - **Functionality;** Operates effectively as it injects an *address* type variable to the *_addressSet* struct and logs the appropriate array index that the input is located within the *members* array. Handled by a *if*() condition to utilise the *contains*() function to verify if the specified address is not already an active element in the struct, if it is the state is not altered. Short address results with no effect to functionality.
  - **Extendability (*Low*);** Implement *bool* conditioning.
- Line 103 - *function remove*()
  - **Speculation;** A function to remove an existing element from the array, it is handled with the *true* return of the *contain*() function to verify that the address is actually consisting within the structure if returned *false* no operation is executed. Firstly the function calls the addresses index that wishes to be removed from the associated key mapping and specifies the last element of the address array using the *length*() function. Then the last element takes the previous submissions index then the array length is decremented, which in theory deletes the last element of the array. The index replacement then follows for the last elements new position and finally a delete operation of the removed addresses index.

        - **Functionality;** Removes an active element if the $if()$ condition is met by validating it's presence in the $\_addressSet$ struct using the $contains()$ function, if the condition is not met it terminates without any further execution. Short emits behaviour as normal.
        - **Extendability (*Low*);** Implement *bool* conditioning would illustrate the completion of the operation, if needing to handle the functionality accordingly.

- Line 125 - $interface\ Voting$
  - A proxy acting $contract\ interface$ to communicate with the specified voting contract.
    - Line 126 - $function\ initiateNomination()$
      - **Speculation;** Straightforward *external* function to return a *bool* dependent on whether a specified delegate has been proposed for a nomination.
    - Line 127 - $initiateRemoval()$ function
      - **Speculation;** Another rather straightforward *external* function that returns a *bool* on the basis whether the specified nominee has been removed.

- Line 130 - $interface\ HydroToken$
  - A proxy acting $contract\ interface$ to externally call with the specified HYDRO token contract
    - Line 131 - $function\ transfer()$
      - **Speculation;** *external* function to verify if the Functionality of a transfer is confirmed dependent on the returning *bool*.
    - Line 132 - $function\ balanceOf()$
      - **Speculation;** Another *external* function to validate the HYDRO balance of a specified *address* by returning a *uint* amount.

- Line 135 - *contract DecentralizationAmbassadors*
  - Core smart contract for the DA operations
  - **Syntax (*High*);** The file name DA.sol and contract name are uncorrelated, results in a compiling error.
    - Line 137 - Library link *SafeMath*
      - **Speculation;** Link to associate *SafeMath* operations with the *uint* variant.
    - Line 138 - Library link *addressSet*
      - **Speculation;** *addressSet* and it's specification to utilise the internal struct of the library.
    - Line 139 - *nominees _addressSet struct*
      - **Speculation;** Internally specifying an *_addressSet* based storage struct for nominees.
    - Line 140 - *ambassadors _addressSet struct*
      - **Speculation;** Internally specifying an *_addressSet* based storage struct for ambassadors.
    - Line 141 - Keymap *lastPayout* obeying *address => uint*
      - **Speculation;** Internally specifying a key mapping structure to keep a log of the last HYDRO payout for each ambassador.
    - Line 142 - *uint payoutBlockNumber*
      - **Speculation;** An *uint* variable to define the *block.number* when payouts execute.
    - Line 143 - *uint payoutHydroAmount*
      - **Speculation;** An *uint* variable to define the amount of HYDRO for each payout.
    - Line 145 - *address hydroAddress*
      - **Speculation;** An *address* of which the source for the HYDRO ERC20 contract is defined.
    - Line 146 - *address votingAddress*

    - **Speculation;** An *address* for which the origin of the voting contract is demoted with no initial specified value.

- Line 148 - $function\ setVotingAddress()$
    - **Functionality;** Executes as expected, assigns a specified *address* to the global *votingAddress* variable. Short address is not required to evaluate as the specified input is only used as a target and not utilised via key mapping.

- Line 152 - $modifier\ onlyDA()$
    - **Speculation;** A modifier to inhibit the non-ambassadors that attempt to interact with a function, by calling the *addressSet contains*() function

- Line 157 - $function\ nominateAmbassador()$
    - **Speculation;** A public function to nominate ambassadors, which begins with an *require*() condition for the $false$ return of the *contains*() function to ensure that the applicant is not an existing ambassador. Then follows the *insert*() function for the *nominees* struct to include the specified address with the other nominees. An *external* call is then taken to the *voting* interface to specify that a nominee has been introduced for nomination with then a proceeding $InitiateNomination$ event is emitted with the specified address indexed to demote the completion of the operation.
    - **Functionality;** Executes as expected once the *external voting* contract $initiateNomination()function$ is modified to hand a *bool* conditioning, once nominated it is an irreversible process therefore you cannot be re-nominated as the specified address cannot be removed from the nominees struct as if it is attempted results in a *revert*. The $InitiateNomination$ event emits successfully without error. Short address submits as normal without any discrepancies observed.

- Line 168 - $function\ finalizeNominationVoting()$
    - **Speculation;** A public function to finalise voting for a specified $_nominee$, which is handled with a *require*() con-

dition declaring that the call must be from the voting contract. Then a proceeding condition is declared if the provided result is $true$ so that the $_nominee$ address is then inserted into the struct using $insert()$ function to conclude with emitting the $FinalizeNomination$ event to demote success.

  · **Functionality;** Operates without any exploits and successfully inserts the finalised nominee to the $ambassadors$ struct if the *bool* is equal to $true$, the $memberIndices$ within the $insert()$ function works flawlessly and logs the specific indexes correctly. If attempted to submit again and the inputted address is already within the ambassadors struct it will be ignored also the concluding $FinalizeNomination$ event emits as expected. Short address attack causes the function to execute as it should without any anomalies.

∗ Line 175 - $function\ ownerAddAmbassador()$

  · **Speculation;** A $onlyOwner()$ modified function for the owner to manually add an ambassador but only if the condition that there is less than 10 current ambassadors, it then follows the same methodology as the previous function except $msg.sender$ takes the place of the address input.

  · **Functionality;** Functions as desired with the added condition that this function can only be applied if the number of current ambassadors is less than 10 active accounts, $reverts$ otherwise if the $array.length$ is larger to this value or equal to it. $FinalizeNomination$ event emits successfully on completion. Short address gives no error in operation.

∗ Line 182 - $function\ initiateAmbassadorRemoval()$

  · **Speculation;** Consists of a public function to propose a removal on an existing ambassador which is handled with a $require()$ condition to verify the current existence of the specified address in the $ambassadors$ struct using the $insert()$ function. The $voting$ contract is externally defined once again using the *votingAddress* to then continue to confirm the removal within the voting contract by calling on the removal operation with the inputted address. To conclude the *InitiateRemovalevent* is emitted with the address of *msg.sender*.

  · **Functionality;** Operates as expected with the condition that the inputted subject has to be an active element within the *ambassadors* struct, *reverts* otherwise. The *InitiateRemoval* event emits the incorrect address. Short address accounts resulted in no notable discrepancies observed.

  · **Syntax (*Medium*);** The event's indexed value should be the *address _ambassador* variable and not *msg.sender*.

∗ Line 191 - *function finalizeRemovalVoting*()

  · **Speculation;** Is a public function to conclude the removal voting process but is only operable by the *voting* contract as it's handled by a *require*() condition declaring *msg.sender* has to be equal to the *votingAddress*. The function then continues to be handled by a condition that only executes if the *bool* result inputted is *true*, which then follows by the *remove*() function being applied to the proposed subject to be removed from the *ambassadors* struct to conlude with emitting the *FinalizeRemoval* event with the subject indexed.

  · **Functionality;** Condition handling operates as desired allowing the active account's removal from the *ambassadors* struct if the *bool* value is equal to *true* but also reverts if the specified account is not engaged in the active role or the caller is not that of the *voting* contract. The *FinalizeRemoval* event emits without error. Short address bares no mis-functions.

∗ Line 199 - *function selfRemoval*()

  · **Speculation;** A public function that is handled with the *onlyDA* modifier that inhibits any non-ambassadors interacting with the function, the *msg.sender* is removed from the struct by utilising the *remove*() function and then concludes with emitting the *FinalizeRemoval* with *msg.sender* as the indexed value.

- **Functionality;** Operates as expected and reverts if the *msg.sender* is not an active ambassador, *FinalizeRemoval* emits without fail. Calling as a short address *msg.sender* resulted in no dysfunctional behaviour.

∗ Line 204 - *function recieveHydro*()

- **Speculation;** Another *public* function that is defined to be only operable by obeying the *onlyDA*() modifier in order to receive the HYDRO dividends for their participation as the specialised role. Which begins with an *external* definition of the HYDRO ERC20 contract by specifying the *hydroAddress* and wrapping around the *HydroToken* interface to declare the contract variable. To follow is a *require*() condition that states that the HYDRO *balanceOf*() of the contract must be greater or equal than the *payoutHydroAmount*. To proceed a variable to specify the last *block.number* payout is defined by utilising the key mapping for *lastPayout*, with then an *if*() condition declaring that the *block.number* must be greater than initial payout block plus the last block payout for the operation to continue. Another *if*() condition is specified to validate if the *transfer*() operation returns a successful execution, which would correlate to a *true bool* result. If obeyed the previous payout block plus the *payoutBlockNumber* are stored within the *lastPayout* keymap, with then to conclude with the emit of the *Payout* event indexing the payee's address and the *uint* amount delivered.
- **Functionality;** This payout based operation works effectively, the previous log to the *lastPayout* keymap is accurate and operates effectively which allows the two *if*() conditions handled within the function to inhibit any early payouts or accounts who are not active ambassadors. The *Payout* event emits the indexed arguments as expected with no error. Short address as operating via *msg.sender* returned no inoperable behaviour.

- **Security (*Medium*);** Potential exploitation of an re-enterancy attack due to the ordering of operations leave the function susceptible. Following the methodology of Checks Effects Interactions, this can be avoided by stating internal contract changes first then to finalise with you're external call.
- **Solution;**

```
function recieveHydro() public onlyDA {
       HydroToken hydro = HydroToken(hydroAddress);
       uint daLastPayoutBlock = lastPayout[msg.sender];

       if (daLastPayoutBlock + payoutBlockNumber < block.number){
        emit Payout(msg.sender, payoutHydroAmount);
        lastPayout[msg.sender] = daLastPayoutBlock + payoutBlockNumber;
        require(hydro.balanceOf(this) >= payoutHydroAmount);
        require(hydro.transfer(msg.sender, payoutHydroAmount));
        }
      }
```

- Manticore and Mythril Findings
  - Solium
    - Syntax warnings and indentation only detected.
  - Mythril
    - Analysis resulted in no alerts of any anomalies or security warnings.
  - Manticore
    - 37 Symbolic Functionality test-cases conducted
      - 34 of the attempted attacks produced an *REVERT*
      - 3 of the attempted attacks produced an *RETURN*
      - **Attack 1;** detected an uninitialised storage in the *addressSet* library, which could cause problems of contract variables overwriting one another but since this is a library and is the definition of a variable it shouldn't be a concern.
      - **Attack 2;** was actually the result of a *TXERROR* return
      - **Attack 3;** unknown could not find the log files.