GetSmart

WizRefund Smart Contract Audit

If you have any questions about smart-contract audit,
contact us hello@getsmart.site

10.12.2020 www.getsmart.site

TABLE OF CONTENTS

INTRODUCTION

Our company provides comprehensive, independent smart contract auditing. We help stakeholders confirm the quality and security of their smart contracts using our standardized audit process. The scope of this audit was to analyze and document the WizRefund contract.

AUDIT METHODOLOGY

WizRefund contract audit consist of three categories of analysis.

1. Design Patterns

We inspect the structure of the smart contract, including both manual and automated analysis.

2. Static Analysis

The static analysis is performed using a series of automated tools, purposefully designed to test the security of the contract.

All the issues found by tools were manually checked (rejected or confirmed).

3. Manual Analysis

Contract reviewing to identify common vulnerabilities. Comparing of requirements and implementation. Reviewing of a smart contract for compliance with specified customer requirements. Checking for a gas optimization and self-documentation. Running tests of the properties of the smart contract in test net.

4. Contracts Reviewed

On December 17, 2020, the source code of the smart contract was reviewed, located in Appendix 1.

This contract is intended to burn “WIZ” tokens in exchange for “ETH”. The contract with “WIZ” tokens is located here:
https://etherscan.io/address/0x2f9b6779c37df5707249eeb3734bbfc94763fbe2
The tokens are currently distributed across 2018 wallets.

ISSUES DISCOVERED

Issues are listed from most critical to least critical. Severity is determined by an assessment of the risk of exploitation or otherwise unsafe behavior.

Severity Levels

- Critical - Funds may be allocated incorrectly, lost or otherwise result in a significant loss.
- High - Affects the ability of the contract to work as designed in a significant way.
- Medium - Affects the ability of the contract to operate.
- Low - Minimal impact on operational ability.
- Informational - No impact on the contract.

AUDIT SUMMARY

Analysis Results

The summary result of the audit performed is presented in the table below

Findings list

Level	Amount
Critical	0
High	0
Medium	0
Low	0
Informational	0

ISSUES

Critical

Not found

High

Not found

Medium

Not found

Low

Not found

Informational

Not found

CONCLUSION

The source code of the smart contract has been thoroughly tested and no vulnerabilities have been found.

All arithmetic operations are performed using the Safe Math Library. Functions that transfer ETH are protected from Reentrancy attacks.

The code uses the world's best practices from the https://github.com/OpenZeppelin/openzeppelin-contracts/ website.

This smart contract can be installed on the Mainnet.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity >=0.6.0 <0.8.0;
3
4  /**
5   * @dev Wrappers over Solidity's arithmetic operations with added overflow
6   * checks.
7   *
8   * Arithmetic operations in Solidity wrap on overflow. This can easily result
9   * in bugs, because programmers usually assume that an overflow raises an
10  * error, which is the standard behavior in high level programming languages.
11  * `SafeMath` restores this intuition by reverting the transaction when an
12  * operation overflows.
13  *
14  * Using this library instead of the unchecked operations eliminates an entire
15  * class of bugs, so it's recommended to use it always.
16  */
17 library SafeMath {
18     /**
19      * @dev Returns the addition of two unsigned integers, reverting on
20      * overflow.
21      *
22      * Counterpart to Solidity's `+` operator.
23      *
24      * Requirements:
25      * - Addition cannot overflow.
26      */
27     function add(uint256 a, uint256 b) internal pure returns (uint256) {
28         uint256 c = a + b;
29         require(c >= a, "SafeMath: addition overflow");
30
31         return c;
32     }
33
34     /**
35      * @dev Returns the subtraction of two unsigned integers, reverting on
36      * overflow (when the result is negative).
37      *
38      * Counterpart to Solidity's `-` operator.
```

```
39          *
40          * Requirements:
41          * - Subtraction cannot overflow.
42          */
43         function sub(uint256 a, uint256 b) internal pure returns (uint256) {
44             return sub(a, b, "SafeMath: subtraction overflow");
45         }
46
47         /**
48          * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
49          * overflow (when the result is negative).
50          *
51          * Counterpart to Solidity's `-` operator.
52          *
53          * Requirements:
54          * - Subtraction cannot overflow.
55          *
56          * _Available since v2.4.0._
57          */
58         function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
59             require(b <= a, errorMessage);
60             uint256 c = a - b;
61
62             return c;
63         }
64
65         /**
66          * @dev Returns the multiplication of two unsigned integers, reverting on
67          * overflow.
68          *
69          * Counterpart to Solidity's `*` operator.
70          *
71          * Requirements:
72          * - Multiplication cannot overflow.
73          */
74         function mul(uint256 a, uint256 b) internal pure returns (uint256) {
75             // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
76             // benefit is lost if 'b' is also tested.
77             // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
78             if (a == 0) {
79                 return 0;
80             }
```

```
81
82          uint256 c = a * b;
83          require(c / a == b, "SafeMath: multiplication overflow");
84
85          return c;
86      }
87
88      /**
89       * @dev Returns the integer division of two unsigned integers. Reverts on
90       * division by zero. The result is rounded towards zero.
91       *
92       * Counterpart to Solidity's `/` operator. Note: this function uses a
93       * `revert` opcode (which leaves remaining gas untouched) while Solidity
94       * uses an invalid opcode to revert (consuming all remaining gas).
95       *
96       * Requirements:
97       * - The divisor cannot be zero.
98       */
99      function div(uint256 a, uint256 b) internal pure returns (uint256) {
100         return div(a, b, "SafeMath: division by zero");
101     }
102
103     /**
104      * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
105      * division by zero. The result is rounded towards zero.
106      *
107      * Counterpart to Solidity's `/` operator. Note: this function uses a
108      * `revert` opcode (which leaves remaining gas untouched) while Solidity
109      * uses an invalid opcode to revert (consuming all remaining gas).
110      *
111      * Requirements:
112      * - The divisor cannot be zero.
113      *
114      * _Available since v2.4.0._
115      */
116     function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
117         // Solidity only automatically asserts when dividing by 0
118         require(b > 0, errorMessage);
119         uint256 c = a / b;
120         // assert(a == b * c + a % b); // There is no case in which this doesn't hold
121
122         return c;
```

```
123         }
124
125         /**
126          * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
127          * Reverts when dividing by zero.
128          *
129          * Counterpart to Solidity's `%` operator. This function uses a `revert`
130          * opcode (which leaves remaining gas untouched) while Solidity uses an
131          * invalid opcode to revert (consuming all remaining gas).
132          *
133          * Requirements:
134          * - The divisor cannot be zero.
135          */
136         function mod(uint256 a, uint256 b) internal pure returns (uint256) {
137             return mod(a, b, "SafeMath: modulo by zero");
138         }
139
140         /**
141          * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
142          * Reverts with custom message when dividing by zero.
143          *
144          * Counterpart to Solidity's `%` operator. This function uses a `revert`
145          * opcode (which leaves remaining gas untouched) while Solidity uses an
146          * invalid opcode to revert (consuming all remaining gas).
147          *
148          * Requirements:
149          * - The divisor cannot be zero.
150          *
151          *  Available since v2.4.0.
152          */
153         function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
154             require(b != 0, errorMessage);
155             return a % b;
156         }
157     }
158
159
160     /*
161      * @dev Provides information about the current execution context, including the
162      * sender of the transaction and its data. While these are generally available
163      * via msg.sender and msg.data, they should not be accessed in such a direct
164      * manner, since when dealing with GSN meta-transactions the account sending and
```

```
165     * paying for execution may not be the actual sender (as far as an application
166     * is concerned).
167     *
168     * This contract is only required for intermediate, library-like contracts.
169     */
170   contract Context {
171       // Empty internal constructor, to prevent people from mistakenly deploying
172       // an instance of this contract, which should be used via inheritance.
173       constructor () {}
174       // solhint-disable-previous-line no-empty-blocks
175
176       function _msgSender() internal view returns (address payable) {
177           return msg.sender;
178       }
179
180       function  msgData() internal view returns (bytes memory) {
181           this;
182           // silence state mutability warning without generating bytecode - see
183   https://github.com/ethereum/solidity/issues/2691
184           return msg.data;
185       }
186   }
187
188   /**
189    * @dev Contract module which provides a basic access control mechanism, where
190    * there is an account (an owner) that can be granted exclusive access to
191    * specific functions.
192    *
193    * This module is used through inheritance. It will make available the modifier
194    * `onlyOwner`, which can be applied to your functions to restrict their use to
195    * the owner.
196    */
197   contract Ownable is Context {
198       address private _owner;
199
200       event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
201
202       /**
203        * @dev Initializes the contract setting the deployer as the initial owner.
204        */
205       constructor () {
206           address msgSender = _msgSender();
```

```
207         _owner = msgSender;
208         emit OwnershipTransferred(address(0), msgSender);
209     }
210
211     /**
212      * @dev Returns the address of the current owner.
213      */
214     function owner() public view returns (address) {
215         return _owner;
216     }
217
218     /**
219      * @dev Throws if called by any account other than the owner.
220      */
221     modifier onlyOwner() {
222         require(isOwner(), "Ownable: caller is not the owner");
223         _;
224     }
225
226     /**
227      * @dev Returns true if the caller is the current owner.
228      */
229     function isOwner() public view returns (bool) {
230         return _msgSender() == _owner;
231     }
232
233     /**
234      * @dev Leaves the contract without owner. It will not be possible to call
235      * `onlyOwner` functions anymore. Can only be called by the current owner.
236      *
237      * NOTE: Renouncing ownership will leave the contract without an owner,
238      * thereby removing any functionality that is only available to the owner.
239      */
240     function renounceOwnership() public onlyOwner {
241         emit OwnershipTransferred(_owner, address(0));
242         _owner = address(0);
243     }
244
245     /**
246      * @dev Transfers ownership of the contract to a new account (`newOwner`).
247      * Can only be called by the current owner.
248      */
```

```
249         function transferOwnership(address newOwner) public onlyOwner {
250             _transferOwnership(newOwner);
251         }
252 
253         /**
254          * @dev Transfers ownership of the contract to a new account (`newOwner`).
255          */
256         function _transferOwnership(address newOwner) internal {
257             require(newOwner != address(0), "Ownable: new owner is the zero address");
258             emit OwnershipTransferred(_owner, newOwner);
259             _owner = newOwner;
260         }
261     }
262 
263     /**
264      * @title Roles
265      * @dev Library for managing addresses assigned to a Role.
266      */
267     library Roles {
268         struct Role {
269             mapping(address => bool) bearer;
270         }
271 
272         /**
273          * @dev Give an account access to this role.
274          */
275         function add(Role storage role, address account) internal {
276             require(!has(role, account), "Roles: account already has role");
277             role.bearer[account] = true;
278         }
279 
280         /**
281          * @dev Remove an account's access to this role.
282          */
283         function remove(Role storage role, address account) internal {
284             require(has(role, account), "Roles: account does not have role");
285             role.bearer[account] = false;
286         }
287 
288         /**
289          * @dev Check if an account has this role.
290          * @return bool
```

```
291          */
292         function has(Role storage role, address account) internal view returns (bool) {
293             require(account != address(0), "Roles: account is the zero address");
294             return role.bearer[account];
295         }
296     }
297
298     /**
299      * @dev Contract module that helps prevent reentrant calls to a function.
300      *
301      * Inheriting from `ReentrancyGuard` will make the {nonReentrant} modifier
302      * available, which can be applied to functions to make sure there are no nested
303      * (reentrant) calls to them.
304      *
305      * Note that because there is a single `nonReentrant` guard, functions marked as
306      * `nonReentrant` may not call one another. This can be worked around by making
307      * those functions `private`, and then adding `external` `nonReentrant` entry
308      * points to them.
309      *
310      * TIP: If you would like to learn more about reentrancy and alternative ways
311      * to protect against it, check out our blog post
312      * https://blog.openzeppelin.com/reentrancy-after-istanbul/[Reentrancy After Istanbul].
313      */
314     abstract contract ReentrancyGuard {
315         // Booleans are more expensive than uint256 or any type that takes up a full
316         // word because each write operation emits an extra SLOAD to first read the
317         // slot's contents, replace the bits taken up by the boolean, and then write
318         // back. This is the compiler's defense against contract upgrades and
319         // pointer aliasing, and it cannot be disabled.
320
321         // The values being non-zero value makes deployment a bit more expensive,
322         // but in exchange the refund on every call to nonReentrant will be lower in
323         // amount. Since refunds are capped to a percentage of the total
324         // transaction's gas, it is best to keep them low in cases like this one, to
325         // increase the likelihood of the full refund coming into effect.
326         uint256 private constant _NOT_ENTERED = 1;
327         uint256 private constant _ENTERED = 2;
328
329         uint256 private _status;
330
331         constructor () {
332             _status = _NOT_ENTERED;
```

```
333         }
334
335         /**
336          * @dev Prevents a contract from calling itself, directly or indirectly.
337          * Calling a `nonReentrant` function from another `nonReentrant`
338          * function is not supported. It is possible to prevent this from happening
339          * by making the `nonReentrant` function external, and make it call a
340          * `private` function that does the actual work.
341          */
342         modifier nonReentrant() {
343             // On the first call to nonReentrant,  notEntered will be true
344             require(_status != _ENTERED, "ReentrancyGuard: reentrant call");
345
346             // Any calls to nonReentrant after this point will fail
347             _status = _ENTERED;
348
349             _;
350
351             // By storing the original value once again, a refund is triggered (see
352             // https://eips.ethereum.org/EIPS/eip-2200)
353             _status = _NOT_ENTERED;
354         }
355     }
356
357     abstract contract Token_interface {
358         function owner() public view virtual returns (address);
359
360         function decimals() public view virtual returns (uint8);
361
362         function balanceOf(address who) public view virtual returns (uint256);
363
364         function transfer(address _to, uint256 _value) public virtual returns (bool);
365
366         function allowance(address _owner, address _spender) public virtual returns (uint);
367
368         function transferFrom(address _from, address _to, uint _value) public virtual returns (bool);
369     }
370
371     /**
372      * @title TokenRecover
373      * @author Vittorio Minacori (https://github.com/vittominacori)
374      * @dev Allow to recover any ERC20 sent into the contract for error
```

```
375   */
376  contract TokenRecover is Ownable {
377
378      /**
379       * @dev Remember that only owner can call so be careful when use on contracts generated from other contracts.
380       * @param tokenAddress The token contract address
381       * @param tokenAmount Number of tokens to be sent
382       */
383      function recoverERC20(address tokenAddress, uint256 tokenAmount) external onlyOwner {
384          Token_interface(tokenAddress).transfer(owner(), tokenAmount);
385      }
386  }
387
388  contract AdminRole is Context, Ownable, ReentrancyGuard {
389      using Roles for Roles.Role;
390      using SafeMath for uint256;
391
392      Roles.Role private _admins;
393      address[] private _signatures;
394
395      constructor () {
396          _admins.add(address(0x8186a47C412f8112643381EAa3272a66973E32f2));
397          _admins.add(address(0xEe3EA17E0Ed56a794e9bAE6F7A6c6b43b93333F5));
398      }
399
400      modifier onlyAdmin() {
401          require(isAdmin(_msgSender()), "AdminRole: you don't have permission to perform that action");
402          _;
403      }
404
405      modifier onlyOwnerOrAdmin() {
406          require(isAdminOrOwner(_msgSender()), "you don't have permission to perform that action");
407          _;
408      }
409
410      function isAdminOrOwner(address account) public view returns (bool) {
411          return isAdmin(account) || isOwner();
412      }
413
414      function isAdmin(address account) public view returns (bool) {
415          return _admins.has(account);
416      }
```

```
417
418         //adding a signature for the next operation
419         function addSignature4NextOperation() public onlyOwnerOrAdmin {
420             bool exist = false;
421             for (uint256 i = 0; i < _signatures.length; i++) {
422                 if (_signatures[i] == _msgSender()) {
423                     exist = true;
424                     break;
425                 }
426             }
427             require(!exist, "You signature already exists");
428             _signatures.push(_msgSender());
429         }
430
431         // removing a signature for the next operation
432         function cancelSignature4NextOperation() public onlyOwnerOrAdmin {
433             for (uint256 i = 0; i < _signatures.length; i++) {
434                 if (_signatures[i] == _msgSender()) {
435                     _remove_signatures(i);
436                     return;
437                 }
438             }
439             require(false, "not found");
440
441         }
442
443         function checkValidMultiSignatures() public view returns (bool){
444             return _signatures.length >= 2;
445             //all signatures = 3 (1 for owner + 2 for admin)
446         }
447
448         function revokeAllMultiSignatures() public onlyOwnerOrAdmin {
449             delete _signatures;
450         }
451
452         function checkExistSignature(address account) public view returns (bool){
453             bool exist = false;
454             for (uint256 i = 0; i < _signatures.length; i++) {
455                 if (_signatures[i] == account) {
456                     exist = true;
457                     break;
458                 }
```

```
459             }
460             return exist;
461         }
462
463         function _remove_signatures(uint index) private {
464             if (index >= _signatures.length) return;
465             for (uint i = index; i < _signatures.length - 1; i++) {
466                 _signatures[i] = _signatures[i + 1];
467             }
468             _signatures.pop();
469         }
470
471     }
472
473
474     contract WisRefund is AdminRole, TokenRecover {
475         using SafeMath for uint256;
476
477         uint256 constant PHASES_COUNT = 4;
478         uint256 private _token_exchange_rate = 273789679021000; //0.000273789679021 ETH per 1 token
479         uint256 private _totalburnt = 0;
480
481         address payable[] private _participants;
482
483         mapping(address => uint256) private _burnt_amounts;
484         mapping(address => bool) private _participants_with_request;
485         mapping(address => bool) private _is_final_withdraw;
486
487         struct PhaseParams {
488             string NAME;
489             bool IS_STARTED;
490             bool IS_FINISHED;
491         }
492         PhaseParams[] public phases;
493
494         Token_interface public token;
495
496         event BurningRequiredValues(uint256 allowed_value, uint256 topay_value, address indexed sc_address, uint256
497     sc_balance);
498         event WithdrawETH(address indexed wallet, uint256 amount);
499         event RefundValue(address indexed wallet, uint256 amount);
500
```

```
501         constructor () {
502
503             token = Token_interface(address(0x2F9b6779c37DF5707249eEb3734BbfC94763fBE2));
504
505             // 0 - first
506             PhaseParams memory phaseInitialize;
507             phaseInitialize.NAME = "Initialize";
508             phaseInitialize.IS_STARTED = true;
509             phases.push(phaseInitialize);
510
511             // 1 - second
512             // tokens exchanging is active in this phase, tokenholders may burn their tokens using
513             // one of the following methods:
514             // method 1: tokenholder has to call approve(params: this SC address, amount in
515             //           uint256) method in Token SC, then he/she has to call refund()
516             //           method in this SC, all tokens from amount will be exchanged and the
517             //           tokenholder will receive his/her own ETH on his/her own address
518             // method 2: tokenholder has to call approve(params: this SC address, amount in
519             //           uint256) method in Token SC, then he/she has to call
520             //           refundValue(amount in uint256) method in this SC, all tokens
521             //           from the refundValue's amount field will be exchanged and the
522             //           tokenholder will receive his/her own ETH on his/her own address
523             // method 3: if somebody accidentally sends tokens to this SC directly you may use
524             //           refundTokensTransferredDirectly(params: tokenholder ETH address, amount in
525             //           uint256) method with mandatory multisignatures
526             PhaseParams memory phaseFirst;
527             phaseFirst.NAME = "the First Phase";
528             phases.push(phaseFirst);
529
530             // 2 - third
531             // in this phase tokeholders who exchanged their own tokens in phase 1 may claim a
532             // remaining ETH stake with register() method
533             PhaseParams memory phaseSecond;
534             phaseSecond.NAME = "the Second Phase";
535             phases.push(phaseSecond);
536
537             // 3 - last
538             // this is a final distribution phase. Everyone who left the request during the
539             // phase 2 with register() method will get remaining ETH amount
540             // in proportion to their exchanged tokens
541             PhaseParams memory phaseFinal;
542             phaseFinal.NAME = "Final";
```

```
543         phases.push(phaseFinal);
544     }
545
546     //
547     // ###################################
548     //
549
550     //only owner or admins can top up the smart contract with ETH
551     receive() external payable {
552         require(isAdminOrOwner(_msgSender()), "the contract can't receive amount from this address");
553     }
554
555     // owner or admin may withdraw ETH from this SC, multisig is mandatory
556     function withdrawETH(address payable recipient, uint256 value) external onlyOwnerOrAdmin nonReentrant {
557         require(checkValidMultiSignatures(), "multisig is mandatory");
558         require(address(this).balance >= value, "not enough funds");
559         (bool success,) = recipient.call{value : value}("");
560         require(success, "Transfer failed");
561         emit WithdrawETH(msg.sender, value);
562         revokeAllMultiSignatures();
563     }
564
565     function setExchangeRate(uint256 _new_value) onlyOwnerOrAdmin external returns (bool){
566         bool result;
567         if (_new_value > 0) {
568             _token_exchange_rate = _new_value;
569             result = true;
570         }
571         return result;
572     }
573
574     function getExchangeRate() external view returns (uint256){
575         return _token_exchange_rate;
576     }
577
578     function getBurntAmountByAddress(address holder) public view returns (uint256){
579         return _burnt_amounts[holder];
580     }
581
582     function getBurntAmountTotal() external view returns (uint256) {
583         return _totalburnt;
584     }
```

```
585
586         function getParticipantAddressByIndex(uint256 index) external view returns (address){
587             return _participants[index];
588         }
589
590         function getNumberOfParticipants() external view returns (uint256){
591             return _participants.length;
592         }
593
594         function getRegistrationStatus(address participant) external view returns (bool){
595             return _participants_with_request[participant];
596         }
597
598         //
599         // ###################################
600         //
601         // tokenholder has to call approve(params: this SC address, amount in uint256)
602         // method in Token SC, then he/she has to call refund() method in this
603         // SC, all tokens from amount will be exchanged and the tokenholder will receive
604         // his/her own ETH on his/her own address
605         function refund() external {
606             address sender = _msgSender();
607             uint256 allowed_value = token.allowance(sender, address(this));
608             refundValue(allowed_value);
609         }
610         // tokenholder has to call approve(params: this SC address, amount in uint256)
611         // method in Token SC, then he/she has to call refundValue(amount in
612         // uint256) method in this SC, all tokens from the refundValue's amount
613         // field will be exchanged and the tokenholder will receive his/her own ETH on his/her
614         // own address
615         function refundValue(uint256 value) public nonReentrant {
616             uint256 i = getCurrentPhaseIndex();
617             require(i == 1 && !phases[i].IS_FINISHED, "Not Allowed phase");
618             // First phase
619
620             address payable sender = _msgSender();
621             uint256 allowed_value = token.allowance(sender, address(this));
622             bool is_allowed = allowed_value >= value;
623
624             require(is_allowed, "Not Allowed value");
625
626             uint256 topay_value = value.mul(_token_exchange_rate).div(10 ** 18);
```

```
627             BurningRequiredValues(allowed_value, topay_value, address(this), address(this).balance);
628             require(address(this).balance >= topay_value, "Insufficient funds");
629 
630             require(token.transferFrom(sender, address(0), value), "Error with transferFrom");
631 
632             if (_burnt_amounts[sender] == 0) {
633                 _participants.push(sender);
634             }
635 
636             _burnt_amounts[sender] = _burnt_amounts[sender].add(value);
637             _totalburnt = _totalburnt.add(value);
638 
639             (bool success,) = sender.call{value : topay_value}("");
640             require(success, "Transfer failed");
641             emit RefundValue(msg.sender, topay_value);
642         }
643 
644         // if somebody accidentally sends tokens to this SC directly you may use
645         // burnTokensTransferredDirectly(params: tokenholder ETH address, amount in
646         // uint256)
647         // requires multisig 2/3
648         function refundTokensTransferredDirectly(address payable participant, uint256 value) external onlyOwnerOrAdmin
649     nonReentrant {
650             uint256 i = getCurrentPhaseIndex();
651             require(i == 1, "Not Allowed phase");
652             // First phase
653 
654             require(checkValidMultiSignatures(), "multisig is mandatory");
655 
656             uint256 topay_value = value.mul(_token_exchange_rate).div(10 ** uint256(token.decimals()));
657             require(address(this).balance >= topay_value, "Insufficient funds");
658 
659             require(token.transfer(address(0), value), "Error with transfer");
660 
661             if (_burnt_amounts[participant] == 0) {
662                 _participants.push(participant);
663             }
664 
665             _burnt_amounts[participant] = _burnt_amounts[participant].add(value);
666             _totalburnt = _totalburnt.add(value);
667 
668             revokeAllMultiSignatures();
```

```
669
670             (bool success,) = participant.call{value : topay_value}("");
671             require(success, "Transfer failed");
672             emit RefundValue(participant, topay_value);
673         }
674
675         // This is a final distribution after phase 2 is fihished, everyone who left the
676         // request with register() method will get remaining ETH amount
677         // in proportion to their exchanged tokens
678         // requires multisig 2/3
679         function startFinalDistribution(uint256 _start_index, uint256 _end_index) external onlyOwnerOrAdmin nonReentrant {
680             require(_end_index < _participants.length);
681             uint256 j = getCurrentPhaseIndex();
682             require(j == 3 && !phases[j].IS_FINISHED, "Not Allowed phase");
683             // Final Phase
684
685             require(checkValidMultiSignatures(), "multisig is mandatory");
686
687             uint256 total_balance = address(this).balance;
688             uint256 sum_burnt_amount = getRefundedAmountByRequests(_start_index, _end_index);
689
690             uint256 pointfix = 1000000000000000000;
691             // 10^18
692
693             for (uint i = _start_index; i <= _end_index; i++) {
694                 uint256 piece = getBurntAmountByAddress(_participants[i]).mul(pointfix).div(sum_burnt_amount);
695                 uint256 value = total_balance.mul(piece).div(pointfix);
696                 if (value > 0 && !isFinalWithdraw(_participants[i])) {
697                     _is_final_withdraw[_participants[i]] = true;
698                     (bool success,) = _participants[i].call{value : value}("");
699                     require(success, "Transfer failed");
700                     emit WithdrawETH(_participants[i], value);
701                 }
702             }
703
704             revokeAllMultiSignatures();
705         }
706
707         function isFinalWithdraw(address _wallet) public view returns (bool) {
708             return _is_final_withdraw[_wallet];
709         }
710
```

```
711         function getRefundedAmountByRequests(uint256 __start_index, uint256 __end_index) public view returns (uint256){
712             require(__end_index < _participants.length);
713             uint256 sum_burnt_amount = 0;
714             for (uint i = __start_index; i <= __end_index; i++) {
715                 if (!isFinalWithdraw(_participants[i])) {
716                     sum_burnt_amount = sum_burnt_amount.add(getBurntAmountByAddress(_participants[i]));
717                 }
718             }
719             return sum_burnt_amount;
720         }
721 
722         // tokeholders who exchanged their own tokens in phase 1 may claim a remaining ETH stake
723         function register() external {
724             _write_register(_msgSender());
725         }
726 
727         // admin can claim register() method instead of tokenholder
728         function forceRegister(address payable participant) external onlyOwnerOrAdmin {
729             _write_register(participant);
730         }
731 
732         function _write_register(address payable participant) private {
733             uint256 i = getCurrentPhaseIndex();
734             require(i == 2 && !phases[i].IS_FINISHED, "Not Allowed phase");
735             // Second phase
736 
737             require(_burnt_amounts[participant] > 0, "This address doesn't have exchanged tokens");
738 
739             _participants_with_request[participant] = true;
740         }
741 
742         function startNextPhase() external onlyOwnerOrAdmin {
743             uint256 i = getCurrentPhaseIndex();
744             require((i + 1) < PHASES_COUNT);
745             require(phases[i].IS_FINISHED);
746             phases[i + 1].IS_STARTED = true;
747         }
748 
749         function finishCurrentPhase() external onlyOwnerOrAdmin {
750             uint256 i = getCurrentPhaseIndex();
751             phases[i].IS_FINISHED = true;
752         }
```

```
753
754         // this method reverts the current phase to the previous one
755         function revertPhase() external onlyOwnerOrAdmin {
756             uint256 i = getCurrentPhaseIndex();
757
758             require(i > 0, "Initialize phase is active already");
759
760             phases[i].IS_STARTED = false;
761             phases[i].IS_FINISHED = false;
762
763             phases[i - 1].IS_STARTED = true;
764             phases[i - 1].IS_FINISHED = false;
765         }
766
767         function getPhaseName() external view returns (string memory){
768             uint256 i = getCurrentPhaseIndex();
769             return phases[i].NAME;
770         }
771
772         function getCurrentPhaseIndex() public view returns (uint256){
773             uint256 current_phase = 0;
774             for (uint256 i = 0; i < PHASES_COUNT; i++)
775             {
776                 if (phases[i].IS_STARTED) {
777                     current_phase = i;
778                 }
779             }
780             return current_phase;
781         }
782     }
```