HAECHI AUDIT

Decubate

Smart Contract Security Analysis

Published on : Oct 16, 2021

Version v2.0

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

HAECHI AUDIT

Smart Contract Audit Certificate

Decubate

Security Report Published by HAECHI AUDIT
v1.0 Oct 16, 2021
v2.0 Oct 26, 2021

Auditor : Felix Kim

[illegible]

Executive Summary

Severity of Issues	Findings	Resolved	Unresolved	Acknowledged	Comment
Critical	-	-	-	-	-
Major	2	2	-	-	All issues resolved
Minor	5	4	-	-	-
Tips	1	-	-	-	-

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

TABLE OF CONTENTS

7 Issues (0 Critical, 2 Major, 5 Minor) Found

ABOUT US

HAECHI AUDIT believes in the power of cryptocurrency and the next paradigm it will bring.
We have the vision to *empower the next generation of finance*. By providing security and trust in the blockchain industry, we dream of a world where everyone has easy access to blockchain technology.

HAECHI AUDIT is a flagship service of HAECHI LABS, the leader of the global blockchain industry.
HAECHI AUDIT provides specialized and professional smart contract security auditing and development services.

We are a team of experts with years of experience in the blockchain field and have been trusted by 300+ project groups. Our notable partners include Universe,1inch, Klaytn, Badger, etc.

HAECHI AUDIT is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@haechi.io

Website: audit.haechi.io

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

INTRODUCTION

This report was prepared to audit the security of Decubate smart contract created by Decubate team. HAECHI AUDIT focused on whether the smart contract created by Decubate team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the smart contract.

CRITICAL	Critical issues must be resolved as critical flaws that can harm a wide range of users.
MAJOR	Major issues require correction because they either have security problems or are implemented not as intended.
MINOR	Minor issues can potentially cause problems and therefore require correction.
TIPS	Tips issues can improve the code usability or efficiency when corrected.

HAECHI AUDIT recommends Decubate team improve all issues discovered. The following issue explanation uses the format of {file name}#{line number}, {contract name}#{function/variable name} to specify the code. For instance, *Sample.sol:20* points to the 20th line of Sample.sol file, while *Sample#fallback()* means the fallback() function of the Sample contract. Please refer to the Appendix to check all results of the tests conducted for this report.

SUMMARY

The codes used in this Audit can be found at GitHub (https://github.com/Decubate-com/smart-contracts/tree/461e92e0d3fdeb80009863469366814bb0d801cc/contracts).The last commit used in this Audit is "461e92e0d3fdeb80009863469366814bb0d801cc".

Issues	HAECHI AUDIT found 0 critical issues, 2 major issues, and 5 minor issues. There are 0 Tips issues that can improve the code's usability or efficiency upon modification.
Update	[v.2.0] In the new commit 896eff83b3308d08a40c07388c00869d482aede2, 2 major issues and 4 minor issues have been corrected.

Severity	Issue	Status
⚠ **MAJOR**	The BracketAllocation#addBracket() function can be called for an already existing Bracket.	(Found - v1.0) (Resolved - v2.0)
⚠ **MAJOR**	When using the DecubateStaking#unStake function, appropriate staking rewards cannot be obtained.	(Found - v1.0) (Resolved - v2.0)
● **MINOR**	Participation is possible before the Agreement begins.	(Found - v1.0) (Resolved - v2.0)
● **MINOR**	The DecubateCrowdfuding#refund() function does not update user information normally.	(Found - v1.0)
● **MINOR**	The DecubateStaking#set() function can update a non-existent pool.	(Found - v1.0) (Resolved - v2.0)
● **MINOR**	The DecubateTiers#getTierOfUser() function may return an unintended value.	(Found - v1.0) (Resolved - v2.0)
● **MINOR**	The BracketAllocation#setBracket() and BracketAllocation#getBracketUsers() functions can be called for non-existent Bracket.	(Found - v1.0) (Resolved - v2.0)

OVERVIEW

Contracts subject to audit

- DecubateStaking
- DecubateCrowdfunding
- BracketAllocation
- DCBTokenWhitelisted
- DecubateTiers
- DecubateVesting

Decubate Smart contract has the following privileges.

- Owner

The details of the control of each privilege are as follows.

Role	Functions
Owner	*Ownable#renounceOwnership()* *Ownable#transferOwnership()* *DecubateStaking#add()* *DecubateStaking#set()* *DecubateStaking#updateFeeValues()* *DecubateCrowdfunding#setDCBAgreement()* *DecubateCrowdfunding#setWalletStoreAddress()* *DecubateCrowdfunding#setTiersAddress()* *DecubateCrowdfunding#setUsingBrackets()* *DecubateCrowdfunding#transferToken()* *BracketAllocation#addBracket()* *BracketAllocation#setBracket()* *BracketAllocation#setCrowdfundingAddress()* *BracketAllocation#setTiersContract()* *DecubateTiers#addTier()* *DecubateTiers#setTier()* *DecubateTiers#setStakingContract()* *DecubateVesting#addInitialUnlock()* *DecubateVesting#setVestingInfo()* *DecubateVesting#addWhitelist()* *DecubateVesting#setDecubateToken()* *DecubateVesting#revoke()* *DecubateVesting#disable()* *DecubateVesting#transferDCB()*

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

FINDINGS

⚠ **MAJOR**

The BracketAllocation#addBracket() function can be called for an already existing Bracket. (Found - v.1.0) (Resolved - v.2.0)

```
function addBracket(
  uint256 _tierId,
  uint256 _maxCount,
  uint256 _allowedAmount
) external onlyOwner returns (bool) {
  bracketInfo[_tierId].addrList = new address[](0);
  bracketInfo[_tierId].maxCount = _maxCount;
 bracketInfo[_tierId].allowedAmount = _allowedAmount;
  bracketInfo[_tierId].currCount = 0;

  return true;
}
```

[https://github.com/Decubate-com/smart-contracts/blob/master/contracts/BracketAllocation.sol#L55-L66]

Issue

Even when the Bracket corresponding to the tier id received as a parameter from *BracketAllocation#addBracket()* already exists, the corresponding function can be called. In this case, addrList and currCount of the already existing Bracket are initialized. addrList is updated through the *BracketAllocation#setAddress()* function, which is designed to be called only by crowdFunding contract. Thus, it has been confirmed that recovery is difficult if data is initialized.

Recommendation

We recommend adding a statement that checks whether there is a Bracket corresponding to the tier id received as a parameter of *BracketAllocation#addBracket()* function.

Update

[v2.0] - The issue has been resolved by changing the *BracketAllocation#addBracket()* function to push to the bracketInfo structure, instead of the old method where it set bracketInfo after receiving a random tier id as a parameter.

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

⚠ **MAJOR**

When using the DecubateStaking#unStake function, appropriate staking rewards cannot be obtained.

(Found - v.1.0) (Resolved - v.2.0)

```
function unStake(uint8 _pid, uint256 _amount) external returns (bool) {
  User storage user = users[_pid][msg.sender];
  Pool storage pool = poolInfo[_pid];

  require(user.total_invested >= _amount, "You don't have enough funds");

  require(canClaim(_pid, msg.sender), "Stake still in locked state");

  pool.totalDeposit = pool.totalDeposit.sub(_amount);
  user.total_invested = user.total_invested.sub(_amount);
  _claim(_pid, msg.sender);
  safeDCBTransfer(msg.sender, _amount);

  return true;
}
```

[https://github.com/Decubate-com/smart-contracts/blob/461e92e0d3fdeb80009863469366814bb0d801cc/contracts/DecubateMasterChef.sol#L274-L288]

Issue

The *DecubateStaking#unStake()* function withdraws some or all of the tokens staked to a specific pool and obtains staking rewards.

The *DecubateStaking#unStake()* function internally calls the *DecubateStaking#_claim()* function. The *DecubateStaking#_claim()* function calls the *DecubateStaking#_payout()* function, calculating the staking rewards that the calling user will receive.

However, the *DecubateStaking#unStake()* function first decreases the total amount of tokens invested by the user, then calls the *DecubateStaking#_claim()* function. Thus, for the tokens that the user intends to unstaking, rewards cannot be received regardless of the staking period.

In such a case, the same result for the two situations below should be expected, but you will get varying results.

Case 1. After staking 1 million tokens for 1 year, unstaking 500,000 tokens

Case 2. After staking 1 million tokens for 1 year, then unstaking 500,000 tokens after calling the *DecubateStaking#claim()* function

In Case 1, the user can only get staking rewards for the remaining 500,000 tokens, but in Case 2, the user can obtain staking rewards for the entire 1 million tokens.

Recommendation

In the *DecubateStaking#unStake()* function, it is recommended to change the order of the statement that decreases user.total_invested and the part that calls the *DecubateStaking#_claim()*.

Update

[v2.0] - The issue has been resolved by changing the order of the statement that reduces user.total_invested in the *DecubateStaking#unStake()* function and the part that calls *DecubateStaking#_claim()*.

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

MINOR

Participation is possible before the Agreement begins.

(Found - v.1.0) (Resolved - v.2.0)

```
struct AgreementInfo {
  address innovatorWallet;
  uint256 softcap;
  uint256 hardcap;
  uint256 createDate;
  uint256 startDate;
  uint256 endDate;
  IERC20 token;
  uint256 vote;
  uint256 totalInvestFund;
  mapping(address => InvestorInfo) investorList;
}
```

[https://github.com/Decubate-com/smart-contracts/blob/461e92e0d3fdeb80009863469366814bb0d801cc/contracts/DecubateCrowdfunding.sol#L42-L53]

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

```
function fundAgreement(uint256 _investFund)
  external
  override
  nonReentrant
  returns (bool)
{
  /** check if user is verified */
  require(_walletStore.isVerified(msg.sender), "User is not verified");

  /** check if investor is willing to invest any funds */
  require(_investFund > 0, "You cannot invest 0");

  /** check if endDate has already passed */
  require(block.timestamp < dcbAgreement.endDate, "Crowdfunding ended");

  require(
    dcbAgreement.totalInvestFund.add(_investFund) <= dcbAgreement.hardcap,
    "Hardcap already met"
  );

  /** Add to bracket if enabled */
  if (_useBrackets && block.timestamp < dcbAgreement.startDate.add(4 hours)) {
```
[https://github.com/Decubate-com/smart-contracts/blob/461e92e0d3fdeb80009863469366814bb0d801cc/contracts/DecubateCrowdfunding.sol#L265-L286]

Issue

DecubateCrowdfunding#AgreementInfo structure stores information related to agreement. Among them, there is a variable startDate that indicates the start time of the agreement. However, the code related to the end time exists in the *DecubateCrowdfunding#fundAgreement()* function, which participates in agreement, yet there is no syntax that reverts when trying to participate in agreement before the start time.

Recommendation

We recommend adding a require() statement that reverts when the current time is earlier than startDate of agreement.

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

Update

[v2.0] - This issue has been resolved by adding a require() statement that confirms whether block.timestamp is after startDate.

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

MINOR

The DecubateCrowdfuding#refund() function does not update user information normally.

(Found - v.1.0)

```solidity
function refund() external override returns (bool) {
  /** check if user is an investor */
  require(
    dcbAgreement.investorList[msg.sender].wallet == msg.sender,
    "User is not an investor"
  );
  /** check if softcap has already reached */
  require(
    dcbAgreement.totalInvestFund < dcbAgreement.softcap,
    "Softcap already reached"
  );
  /** check if end date have passed or not */
  require(block.timestamp >= dcbAgreement.endDate, "End date not reached");
  uint256 _amount = dcbAgreement.investorList[msg.sender].investAmount;

  /** check if contract have enough balance*/
  require(
    dcbAgreement.token.balanceOf(address(this)) >= _amount,
    "Not enough funds in treasury"
  );
  dcbAgreement.investorList[msg.sender].active = false;
  dcbAgreement.investorList[msg.sender].wallet = address(0);
  dcbAgreement.totalInvestFund = dcbAgreement.totalInvestFund.sub(
    dcbAgreement.investorList[msg.sender].investAmount
  );

  dcbAgreement.token.transfer(msg.sender, _amount);

  return true;
}
```

[https://github.com/Decubate-com/smart-contracts/blob/461e92e0d3fdeb80009863469366814bb0d801cc/contracts/DecubateCrowdfunding.sol#L406-L435]

Issue

The *DecubateCrowdfunding#refund()* function cancels participation in agreement. In other words, most of the information allocated by the *DecubateCrowdfunding#fundAgreement()* function must be initialized.

For instance, a variable like the *DecubateCrowdfunding#_participantCount* means the count participating in the agreement. Thus, when the *DecubateCrowdfunding#refund()* function is called, it is confirmed as the value to be decreased.

Also, the information on bracket/tiers is not properly initialized, so when the user participates in the agreement and then tries to participate again by raising the tier after a refund, the allowedAmount does not change. In the case of allowedAmount, it is a value that varies according to the tier, and it is confirmed as a value that needs to be changed when participating again in the agreement.

Recommendation

It is recommended to add logic that properly initializes the information of participating users when the *DecubateCrowdfunding#refund()* function is called.

Acknowledgement

In the case of *DecubateCrowdfunding#_participantCount*, if you intended to record the number of users who have participated at least once, not the number of users currently participating, no modification is necessary.

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

MINOR

The DecubateStaking#set() function can update a non-existent pool.

(Found - v.1.0) (Resolved - v.2.0)

```
function set(
  uint256 _pid,
  uint16 _apy,
  uint16 _lockPeriodInDays,
  uint256 _endDate,
  uint256 _minContrib,
  uint256 _maxContrib,
  uint256 _hardCap
) public onlyOwner {
  poolInfo[_pid].apy = _apy;
  poolInfo[_pid].lockPeriodInDays = _lockPeriodInDays;
  poolInfo[_pid].endDate = _endDate;
  poolInfo[_pid].minContrib = _minContrib;
  poolInfo[_pid].maxContrib = _maxContrib;
  poolInfo[_pid].hardCap = _hardCap;
}
```

[https://github.com/Decubate-com/smart-contracts/blob/461e92e0d3fdeb80009863469366814bb0d801cc/contracts/DecubateMasterChef.sol#L142-L157]

Issue

The *DecubateStaking#set()* function can update pool information even for a non-existent pool id.

Recommendation

It is advised to add a statement to check whether the pool id input to the *DecubateStaking#set()* is smaller than the current pool length.

Update

[v2.0] - This issue has been resolved by adding a require() statement that checks if the pool id coming in as a parameter actually exists.

MINOR

The DecubateTiers#getTierOfUser() function may return an unintended value.(Found - v.1.0) (Resolved - v.2.0)

```
function getTierOfUser(address addr)
  public
  view
  returns (
    bool flag,
    uint256 pos,
    uint256 multiplier
  )
{
  uint256 len = tierInfo.length;
  uint256 totalDeposit = getTotalDeposit(addr);
  multiplier = 1;

  for (uint256 i = 0; i < len; i++) {
    if (
      totalDeposit >= tierInfo[i].minLimit &&
      totalDeposit <= tierInfo[i].maxLimit
    ) {
      pos = i;
      flag = true;
     break;
    }
  }

  // compounding effect for final bracket
  if (!flag && totalDeposit > tierInfo[len - 1].maxLimit) {
    pos = len - 1;
    flag = true;
    // multiplier is the users total deposit divided by the
    // minimum limit in the tier. For example Diamond tier is
    // 80,0000+ DCB. The max limit of the tier should be set
    // 159,999 DCB and when the limit is passed the compounding
   // effect will be used to find the number of tickets e.g 2
    // for 160,000
    multiplier = totalDeposit / (tierInfo[len - 1].minLimit);
  }
```

```

    return (flag, pos, multiplier);
}
```
[https://github.com/Decubate-com/smart-contracts/blob/master/contracts/DecubateTiers.sol#L127-L165]

Issue

The *DecubateTiers#getTierOfUser()* function determines which tier the user belongs to according to the amount of staking and returns thereof. In reference to the comments, each Tier section is identified as [minLimit, maxLimit).

But the code *DecubateTiers#getTierOfUser()* function logic judges each Tier section as [minLimit, maxLimit]. Thus, when the amount the user staking is equal to the minLimit of any Tier, a Tier one level lower than the Tier is returned.

For example, assume the amount the user staking is 2 and the tier is configured as follows:

Silver: {minLimit: 1, maxLimit: 2, index: 0}

Gold: {minLimit: 2, maxLimit: 3, index: 1}

Then, because the value of 2 satisfies the minLimit or more and less than maxLimit of the Silver section, the *DecubateTiers#getTierOfUser()* function returns the Silver section, not the Gold section, to the user's Tier.

Recommendation

It is recommended to modify the comparison syntax to check minLimit or more and less than maxLimit when checking Tier section.

Update

[v2.0] - This issue has been resolved by correcting the statement to check minLimit or more and less than maxLimit in confirming the Tier interval.

MINOR

The BracketAllocation#setBracket() and BracketAllocation#getBracketUsers() functions can be called for non-existent Bracket.(Found - v.1.0) (Resolved - v.2.0)

```
function setBracket(
  uint256 _tierId,
  uint256 _maxCount,
  uint256 _allowedAmount
) external onlyOwner returns (bool) {
  bracketInfo[_tierId].maxCount = _maxCount;
  bracketInfo[_tierId].allowedAmount = _allowedAmount;
  return true;
}
```

[https://github.com/Decubate-com/smart-contracts/blob/master/contracts/BracketAllocation.sol#L73-L81]

```
function getBracketUsers(uint256 tier)
  external
  view
  returns (address[] memory alist)
{
  return bracketInfo[tier].addrList;
}
```

[https://github.com/Decubate-com/smart-contracts/blob/master/contracts/BracketAllocation.sol#L92-L98]

Issue

Each of the *BracketAllocation#BracketAllocation()* and *BracketAllocation#getBracketUsers()* functions receives a tier id as parameter, and each function can be called even if there is no Bracket that corresponds to this tier id.

Recommendation

We recommend adding a statement that checks whether Bracket corresponding to the tier id input in *BracketAllocation#BracketAllocation()*, *BracketAllocation#getBracketUsers()* exists.

Update

[v2.0] - This issue has been resolved by adding a require() statement that checks whether the Bracket corresponding to the tier id input in *BracketAllocation#BracketAllocation()* exists.

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

💡 **TIPS**

There are missing Events.

(Found - v.1.0)

The following list shows functions with missing Events.

Function	Expected Event	Emitted Event	Omitted Event
DCBTokenWhitelisted#burn()	Transfer, Burn	Transfer	Burn

Without Event, it is difficult to identify in real-time whether correct values are recorded on the blockchain. In this case, it becomes problematic to determine whether the corresponding value has been changed in the application and whether the corresponding function has been called.

Thus, we recommended adding Events corresponding to the change occurring in the function.

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the Main-net. In order to write secure smart contracts, correction of discovered problems and sufficient testing thereof are required.

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

Appendix A. Test Results

The following results show the unit test results covering the key logic of the smart contract subject to the security audit. Parts marked in red are test cases that failed to pass the test due to existing issues.

```
BracketAllocation
  #constructor()
    ✓ tiersContact should be set properly
    ✓ crowdFunding contract should be set properly
  BracketAllocation Spec
    #addBracket()
      ✓ should fail if msg.sender is not owner
      1) should fail if the bracket corresponding to the tierId already exists
      valid case
        ✓ addrList should be initialized
        ✓ currCount should be initialized
        ✓ maxCount should be set properly
        ✓ allowedAmount should be set properly
    #setBracket()
      ✓ should fail if msg.sender is not owner
      2) should fail if the bracket corresponding to the tierId not exists
      valid case
        ✓ maxCount should be set properly
        ✓ allowedAmount should be set properly
    #getBracketUsers()
      3) should fail if the bracket corresponding to the tierId not exists
      valid case
        ✓ should return addrList properly (87ms)
    #getBrackets()
      ✓ should return brackets properly (39ms)
    #setCrowdfundingAddress()
      ✓ should fail if msg.sender is not owner
      valid case
        ✓ crowdFunding should be set properly
    #setTiersContract()
      ✓ should fail if msg.sender is not owner
      valid case
        ✓ tiersContract should be set properly
    #setAddress()
      ✓ should fail if msg.sender is not crowdFunding contract (61ms)
```

```
        ✓ should fail if user already has allocation (88ms)
        ✓ should fail if user does not belong to any bracket (80ms)
        ✓ should fail if bracket already filled (129ms)
        valid case
          ✓ address should be pushed into bracket's addrList (157ms)
          ✓ bracket's currCount should increase (146ms)
          ✓ userAllocation active should be true (118ms)
          ✓ userAllocation allowedAmount should be set properly (without multiplier) (119ms)
          ✓ userAllocation allowedAmount should be set properly (with multiplier) (121ms)

  DCBTokenWhitelisted
    #constructor()
      ✓ should set name properly
      ✓ should set symbol properly
      ✓ should set decimals properly
      ✓ should set initial supply properly
      ✓ owner's isWhitelisted should be set  properly
      ✓ should set isBlackListEnabled properly
      ✓ should set isTimeLockEnabled properly
      ✓ should set startTime properly
      ✓ should set blockSellUntil properly
    ERC20 Spec
      #transfer()
        ✓ should fail if recipient is ZERO_ADDRESS
        ✓ should fail if sender's amount is lower than balance
        ✓ should fail if blacklist enabled and sender is blackListed
        ✓ should fail if blacklist enabled and recipient is blackListed
        ✓ should fail if token is timeLocked and sender or recipient is not whitelisted and current
time is less than startTime (38ms)
        ✓ should fail if token sell is disabled (48ms)
        when succeeded
          ✓ sender's balance should decrease
          ✓ recipient's balance should increase
          ✓ should emit Transfer event
      #transferFrom()
        ✓ should fail if sender is ZERO_ADDRESS
        ✓ should fail if recipient is ZERO_ADDRESS
        ✓ should fail if sender's amount is lower than transfer amount
        ✓ should fail if allowance is lower than transfer amount
        ✓ should fail even if try to transfer sender's token without approval process
        ✓ should fail if blacklist enabled and sender is blackListed
        ✓ should fail if blacklist enabled and recipient is blackListed
        ✓ should fail if token is timeLocked and sender or recipient is not whitelisted and current
time is less than startTime
```

```
    ✓ should fail if token sell is disabled
    when succeeded
      ✓ sender's balance should decrease
      ✓ recipient's balance should increase
      ✓ should emit Transfer event
      ✓ allowance should decrease
      ✓ should emit Approval event
  #approve()
    ✓ should fail if spender is ZERO_ADDRESS
    valid case
      ✓ allowance should set appropriately
      ✓ should emit Approval event
ERC20Burnable spec
  #burn()
    ✓ should fail if msg.sender is not owner
    ✓ should fail if try to burn more than burner's balance
    valid case
      ✓ totalSupply should decrease
      ✓ account's balance should decrease
      ✓ should emit Transfer event
      4) should emit Burn event
DCBTokenWhitelisted Spec
  #lockTokens()
    ✓ should fail if msg.sender is not owner
    ✓ should fail if amount is greater than balance
    when succeeded
      ✓ owner's balance should decrease
      ✓ DEAD_ADDRESS's balance should increase
      ✓ should emit Transfer event
  #whiteList()
    ✓ should fail if msg.sender is not owner
    valid case
      ✓ isWhiteListed should be set properly
      ✓ should emit WhiteListSet Event
  #blackList()
    ✓ should fail if msg.sender is not whiteListed
    valid case
      ✓ isBlackListed should be set properly
      ✓ should emit BlackListSet Event
  #bulkBlackList
    ✓ should fail if msg.sender is not whiteListed
    ✓ should fail if input array is mismatched
    valid case
      ✓ isBlackListed should be set properly
    setBlacklist
      ✓ should fail if msg.sender is not whitelisted
      valid case
        ✓ isBlackListEnabled should be set properly
    setTimeLocked
      ✓ should fail if msg.sender is not whitelisted
      valid case
        ✓ isTimeLockEnabled should be set properly
        ✓ startTime should be set properly
    setBlockSellUntil
      ✓ should fail if msg.sender is not whitelisted
      valid case
        ✓ blockSellUntil should be set properly
    setPairAddress
      ✓ should fail if msg.sender is not whitelisted
      valid case
        ✓ pairAddress should be set properly

DecubateCrowdfunding
  #setDCBAgreement()
    ✓ should fail if msg.sender is not owner
    valid case
      ✓ agreement information update
  #setWalletStoreAddress()
    ✓ should fail if msg.sender is not owner
    valid case
      ✓ wallet address update
  #setTiersAddress()
    ✓ should fail if msg.sender is not owner
    valid case
      ✓ tier address update
  #fundAgreement()
    ✓ should fail if msg.sender is not verified
    ✓ should fail if try to invest zero amount
    ✓ should fail if crowdfunding ended
    ✓ should fail if investment amount exceeds hardcap
    case 1 - bracket disabled
      ✓ should fail if user is not part of a tier
      ✓ should fail if user does not approve agreement contract
      valid case - first fund
        ✓ agreement contract get token
        ✓ participant count should increase
        ✓ should emit InverstorJoin event
      valid case - after first fund
```

```
        ✓ agreement contract get token
    case2 - bracket enabled & limited time sale
      ✓ should fail if amount is greater than allocation (60ms)
      valid case - first fund
        ✓ agreement contract get token
        ✓ participant count should increase
        ✓ should emit InverstorJoin event
      valid case - after first fund
        ✓ agreement contract get token
  #claimInnovatorFund()
    ✓ should fail if endDate not passed
    ✓ should fail if softcap not reached
    ✓ should fail if fund does not have enough token (53ms)
    valid case
      ✓ innovator wallet get token
      ✓ should emit ClaimFund event
  #refund()
    ✓ should fail if user is not an investor
    ✓ should fail if softcap already reached
    ✓ should fail if endDate not passed
    ✓ should fail if fund does not have enough token (56ms)
    valid case
      ✓ total invest amount decrease
      5) participant count should decrease
      ✓ refund request user get token

DecubateStaking
  #add()
    ✓ should fail if msg.sender is not owner
    valid case
      ✓ pool length increase
      ✓ pool information update properly
  #set()
    ✓ should fail if msg.sender is not owner
    6) should fail if try to set non-existent pool
    valid case
      ✓ pool information update properly
  #stake()
    ✓ should fail if stakingContract does not allowed
    ✓ should fail if user does not have enough token balance
    ✓ should fail if try to stake less than minimum contribution
    ✓ should fail if try to stake more than maximum contribution
    ✓ should fail if pool is full
    ✓ should fail if try to stake in disalbed period
    valid case
      ✓ pool total Deposit information update
      ✓ user information with respect to pool update
      ✓ should emit Stake event
  #claim()
    ✓ should fail if reward still in locked state
    valid case
      ✓ user get token
      ✓ fee receiver get token
      ✓ should emit Claim event
  #unStake()
    ✓ should fail if try to unstake more than staking amount
    ✓ should fail if stake still in locked state
    valid case
      7) user get reward/staking token
      8) fee receiver get token
      ✓ should emit Claim event

DecubateTiers
  #constructor()
    ✓ stakingContract should be set properly
  DecubateTiers Spec
    #addTier()
      ✓ should fail if msg.sender is not owner
      valid case
        ✓ tierInfo should be added properly
    #setTier()
      ✓ should fail if msg.sender is not owner
      ✓ should fail if tierId is not exist
      valid case
        ✓ tierInfo should be set properly
    #setStakingContract()
      ✓ should fail if msg.sender is not owner
      valid case
        ✓ stakingContract should be changed properly
    #getTotalDeposit()
      ✓ should return user's total deposit properly (88ms)
    #getTiersLength()
      ✓ should return tierInfo's length properly
    #getTiers()
      ✓ should return tierInfo properly
    #getTierOfUser
      ✓ should return user's tier position properly (without the compounding effect) (153ms)
      9) should return next tier position if totalDeposit is equal to the maxLimit of the tier
      ✓ should return multiplier properly if totalDeposit is greater than highest tier's maxLimit
(83ms)

  DecubateVesting
    #addInitialUnlock()
      ✓ should fail if msg.sender is not owner
      ✓ should fail if VIP already exist
      ✓ vipPools should be set properly
    #claimInitialUnlock()
      ✓ should fail if wallet does not have an initial unlock to claim
      ✓ should fail if unlockAmount is zero (38ms)
      ✓ should fail if unlockTime is greater than current time
      valid case
        ✓ unlockAmount should be zero
        ✓ wallet's decubateToken balance should increase
        ✓ should emit Transfer event
    #setVestingInfo()
      ✓ should fail if msg.sender is not owner
      ✓ should fail if _strategy is zero
      ✓ should fail if vesting option is already existing
      valid case
        ✓ strategy should be set properly
        ✓ cliff should be set properly
        ✓ start should be set properly
        ✓ duration should be set properly
        ✓ revocable should be set properly
        ✓ active should be set properly
    #getVestingInfo()
      ✓ should fail if vesting option is not existing
      ✓ should return vesting info properly
    #addWhitelist()
      ✓ should fail if msg.sender is not owner
      ✓ should fail if vesting option is not existing
      ✓ should fail if whitelist is already available (39ms)
      valid case
        ✓ whitelist should be set properly (38ms)
        ✓ should emit AddWhitelist event
    #getWhitelist()
      ✓ should fail if vesting option is not existing
      ✓ should fail if whitelist is not existing
      ✓ should return whitelist properly
    #setDecubateToken()
      ✓ should fail if msg.sender is not owner
      ✓ _decubateToken should be set properly
    #getDecubateToken()
      ✓ should return decubateToken's addresses properly
    #calculateVestAmount()
      ✓ should fail if vesting option is not exsit
      ✓ should fail if user in not in whitelist
      valid case
        ✓ should return zero if vesting option's cliff is greater than now
        ✓ should return dcbAmount if wallet is revoked (43ms)
        ✓ should return dcbAmount if now is greater than vesting option's start + duration
        ✓ else, should return properly
    #calculateReleasableAmount()
      ✓ should fail if vesting option is not exist
      ✓ should fail if user in not in whitelist
      ✓ should return the amount obtained by subtracting distributedAmount from
calculatedVestAmount (89ms)
    #claimDistribution()
      ✓ should fail if vesting option is not existing
      ✓ should fail if user is not in whitelist
      ✓ should fail if user is disabled from claiming token (43ms)
      ✓ should fail if releaseAmount is zero (44ms)
      valid case
        ✓ wallet's decubateToken balance should increase
        ✓ wallet's distributedAmount should increase
    #revoke()
      ✓ should fail if msg.sender is not owner
      ✓ should fail if vesting option is not existing
      ✓ should fail if user is not in whitelist
      ✓ should fail if vesting option is not revocable
      ✓ should fail if user is already revoked (42ms)
      valid case
        ✓ Wallet should be revoked (50ms)
        ✓ should emit Revoked event
    #disable()
      ✓ should fail if msg.sender is not owner
      ✓ should fail if vesting option does not exist
      ✓ should fail if user is not in whitelist
      ✓ should fail if user is already disabled (51ms)
      valid case
        ✓ Wallet should be disabled (53ms)
        ✓ should emit Disabled event
    #transferDCB()
      ✓ should fail if msg.sender is not owner
      ✓ should increase owner's DCB balance
    #getTotalToken()
      ✓ should return totalToken properly
    #hasWhitelist()
      ✓ should fail if vesting option is not existing
      ✓ should return properly
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contract/					
BracketAllocation.sol	100	100	100	100	
DCBTokenWhitelisted.sol	100	100	100	100	
DecubateMasterChef.sol	100	100	100	100	
DecubateCrowdfunding.sol	100	100	100	100	
DecubateTiers.sol	100	100	100	100	

[Table 1] Test Case Coverage

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved

End of Document

COPYRIGHT 2021. HAECHI AUDIT. all rights reserved