Competitive Security Assessment

Dewhales

Jun 16th, 2023

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

Project Detail

Project Name	Dewhales
Platform & Language	Solidity
Codebase	• https://github.com/CodezukiOden/dewhales-private-sales • audit commit - 96aba39f1afa1fe2f06c7b6a05880e9419e09d6a • final commit - 54423f28532cfe8a21b5ab8a6a88e8fb19efbbf6
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

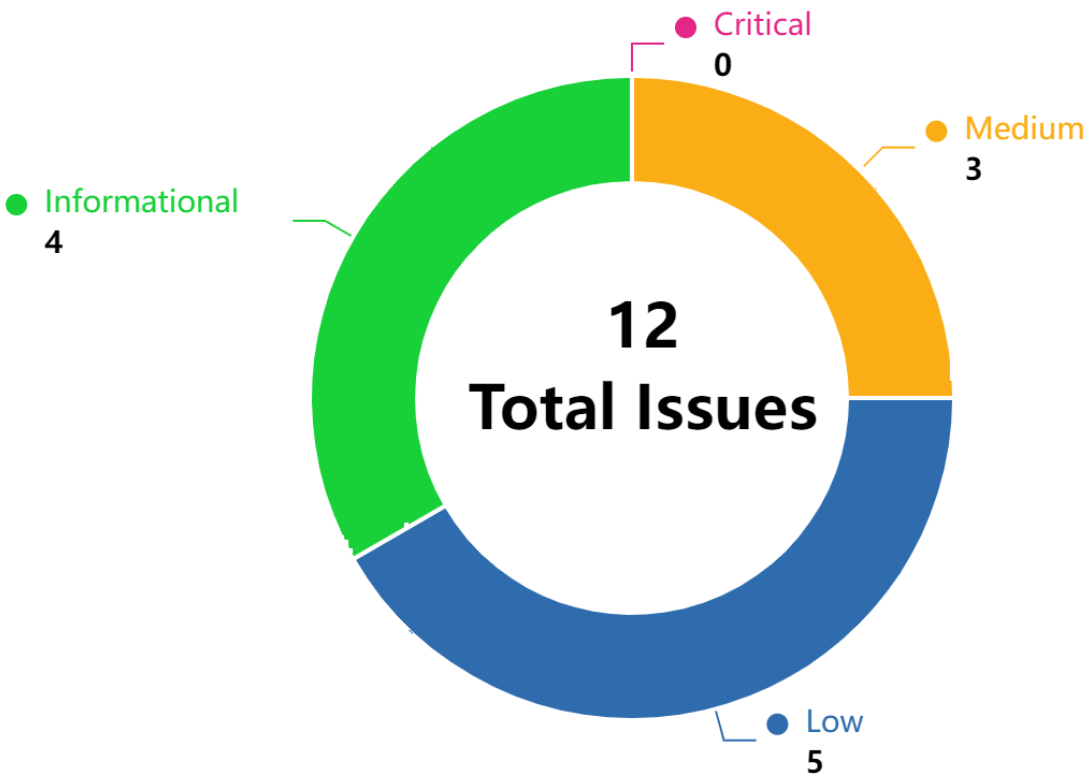
Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	0	0	0	0	0	0
Medium	3	0	0	1	1	1
Low	5	0	3	1	0	1
Informational	4	0	0	3	1	0

Audit Scope

File	Commit Hash
src/PrivateRounds.sol	96aba39f1afa1fe2f06c7b6a05880e9419e09d6a

Code Assessment Findings



ID	Name	Category	Severity	Status	Contributor
DWL-1	tokenAddress for different rounds can be set to same	Privilege Related	Medium	Fixed	parth_15
DWL-2	Wrong calculation of leaf in pledge function	Logical	Medium	Declined	parth_15
DWL-3	function does not check for zero pledges and round existence	Logical	Medium	Mitigated	parth_15, Yaodao, yekong
DWL-4	Floating pragma should not be used	Language Specific	Low	Fixed	parth_15
DWL-5	Incompatibility With Deflationary Tokens	Logical	Low	Acknowledged	Yaodao

DWL-6	Incorrect usage of Ownable constructor	Logical	Low	Declined	yekong
DWL-7	Lack of Necessary Checks in pledge Function Can Lead to Unexpected Behavior and Unnecessary Gas Consumption	Logical	Low	Acknowledged	yekong
DWL-8	Lack of repeatability check implement in the function `setTokenForRound()`	Logical	Low	Acknowledged	Yaodao
DWL-9	Gas Optimization	Gas Optimization	Informational	Mitigated	parth_15, yekong
DWL-10	Unused code	Code Style	Informational	Fixed	Yaodao, yekong
DWL-11	Variables that could be declared as immutable	Code Style	Informational	Fixed	Yaodao
DWL-12	rename the claims mapping to claimed and totalClaims to totalClaimed	Code Style	Informational	Fixed	parth_15

DWL-1: tokenAddress for different rounds can be set to same

Category	Severity	Status	Contributor
Privilege Related	Medium	Fixed	parth_15

Code Reference

- code/src/PrivateRounds.sol#L292
- code/src/PrivateRounds.sol#L299-L301

```
292:        round.tokenAddress = _tokenAddress;

299:        uint256 tokenBalance = IERC20(round.tokenAddress).balanceOf(address(this));
300:        uint256 totalClaimedForRound = totalClaims[_roundId];
301:        uint256 totalTokensReceived = tokenBalance + totalClaimedForRound;
```

Description

parth_15 : It is on total discretion of `owner` to not set `tokenAddress` of round to previously used `token`. If owner by mistakes do it, it will mess a lot of calculations. If this is done, `claimTokens` by `pledgers` won't function as expected which will introduce lot of bugs.

If this is set by mistake, some pledgers of one round will be able to claim more tokens than expected if both of the project rounds send the same tokens to the contract.

Recommendation

parth_15 : Use the `mapping` variable to ensure and check no `tokenAddress` is used twice. Once `tokenAddress` is set using `setTokenForRound`, set the mapping of that token to true so that it shouldn't be used further. Also, before setting `tokenAddress`, the check should be made to ensure that mapping of that token is `false`.

Client Response

Fixed. known, and deemed to be unlikely to happen in any practical scenario (was mentioned in code documentation too)due to ease of mitigation we agree with preventing it though,

DWL-2:Wrong calculation of leaf in pledge function

Category	Severity	Status	Contributor
Logical	Medium	Declined	parth_15

Code Reference

- code/src/PrivateRounds.sol#L203

```
203:        bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(msg.sender, _maxAmount))));
```

Description

parth_15 : The calculation of `leaf` in `pledge` function is done as follows:

```
bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(msg.sender, _maxAmount))));
```

The `bytes.concat` used is irrelevant because there are no multiple bytes to be contacted since `keccak256` returns `bytes32`. Also, the above snippet of code does `keccak256` twice which is not the standard practise and should only be done once.

The calculation done is not standard practise and off-chain client may not be aware of this non-trivial behaviour which can result in failing of transaction.

Recommendation

parth_15 : Consider changing the above snippet of code by removing `bytes.concat` and only doing `keccak256` once.

```
bytes32 leaf = keccak256(abi.encode(msg.sender, _maxAmount));
```

Client Response

Declined. see here https://github.com/OpenZeppelin/merkle-tree this is the library that will also be used on the frontend and has been tested successfully

DWL-3:function does not check for zero pledges and round existence

Category	Severity	Status	Contributor
Logical	Medium	Mitigated	parth_15, Yaodao, yekong

Code Reference

- code/src/PrivateRounds.sol#L126-L132
- code/src/PrivateRounds.sol#L134-L146
- code/src/PrivateRounds.sol#L170-L173
- code/src/PrivateRounds.sol#L175-L184
- code/src/PrivateRounds.sol#L329-L341

```
126:    constructor(address _stablecoinAddress, address _owner, address _feeReceiver) Ownable() {
127:        stablecoinContract = IERC20(_stablecoinAddress);
128:
129:        feeReceiver = _feeReceiver;
130:
131:        _transferOwnership(_owner);
132:    }

134:    function createNewRound(
135:        uint256 _target,
136:        uint256 _groupAllocation,
137:        uint32 _startTime,
138:        uint32 _cappedPeriodEndTime,
139:        uint32 _endTime,
140:        bytes32 _root,
141:        address _projectEOA
142:    ) external onlyOwner {
143:        require(_startTime >= block.timestamp, "start at < now");
144:        require(_cappedPeriodEndTime > _startTime, "cappedEndTime <= startTime");
145:        require(_endTime >= _cappedPeriodEndTime, "endTime < cappedEndTime");
146:        require(_target <= _groupAllocation, "target > groupAllocation");

170:    function changeWhitelistRoot(uint32 _roundId, bytes32 newRoot) external onlyOwner {
171:        rounds[_roundId].whitelistRoot = newRoot;
172:        emit WhitelistRootChanged(_roundId, newRoot);
173:    }
```

```
175:    function cancelRound(
176:        uint32 _roundId
177:    ) external onlyOwner {
178:        Round storage round = rounds[_roundId];
179:
180:        pledgesNotSent(round);
181:
182:        round.cancelled = true;
183:        emit RoundCanceled(_roundId);
184:    }

329:    function receivedAndClaimable(uint256 roundId, address investor) external view returns (uint256, uint2
56){
330:        Round memory round = rounds[roundId];
331:        uint256 tokenBalance = IERC20(round.tokenAddress).balanceOf(address(this));
332:        uint256 totalClaimedForRound = totalClaims[roundId];
333:        uint256 totalTokensReceived = tokenBalance + totalClaimedForRound;
334:
335:        uint pledgedUsdAmount = pledgedAmounts[roundId][investor];
336:
337:        uint tokensClaimable = ((pledgedUsdAmount *
338:            totalTokensReceived) / round.totalUsdPledged) - claims[roundId][investor];
339:
340:        return (totalTokensReceived, tokensClaimable);
341:    }
```

Description

parth_15 : The contract doesn't check the variables `stablecoinContract` and `feeReceiver` are non-zero. Also, while creation of `newRound`, there are no checks on `projectEOA`. Since, this variables are set only once and can't be resetted if something goes wrong, it is needed to put checks on whether they are non-zero to be safe. Also, `changeWhitelistRoot` function doesn't check if `_roundId` it is setting exists or not.

If this variable are set to 0 by mistake, it can't be changed and the contract won't work as intended.

Yaodao : The function `cancelRound()` is used to cancel the round and then refund tokens. The function only checks whether the pledge is sent to the EOA but not checks the whether the token for round is set. As a result, the owner can call this function to cancel the round between the owner set the token for round and the owner call the function `sendPledgesToProject()`. Then the users can call `claimTokens()` to get the project's tokens(if the project has transferred the tokens into the contract) and then call refund to get their amounts back.

Consider below codes:

```
    function cancelRound(
        uint32 _roundId
    ) external onlyOwner {
        Round storage round = rounds[_roundId];

        pledgesNotSent(round);

        round.cancelled = true;
        emit RoundCanceled(_roundId);
    }
```

yekong : Several functions in the contract do not check if the provided addresses are zero addresses. Specifically, the `constructor`, `createNewRound`, and `receivedAndClaimable` functions do not include a zero address check for their address parameters. Using a zero address in these functions could lead to unexpected behavior, such as funds being locked in the contract or rounds being associated with a non-existent project.
yekong : The `changeWhitelistRoot` function does not check if the round specified by `_roundId` exists before changing the whitelist root. This could lead to unexpected behavior as the function may be able to set a whitelist root for a round that does not exist. Additionally, if the round has already started, changing the whitelist root could potentially disrupt the fairness of the round.
yekong : The cancelRound function in the contract lacks necessary checks before setting a round as cancelled. Specifically, it does not verify if the round exists, if the round has already started or ended, or if the round has already been cancelled. This could lead to unpredictable behavior, including the possibility of marking non-existent rounds as cancelled or cancelling a round multiple times.
As the function stands, it could also potentially lock funds if a round is cancelled after participants have made pledges but before pledges are sent.
yekong : The receivedAndClaimable function in the contract, which is meant to calculate and return the total tokens received and the amount claimable by an investor for a given round, does not check if round.totalUsdPledged is zero or if the round exists (i.e., if roundId is valid). This could potentially lead to erroneous results or a division by zero

Recommendation

parth_15 : Impose a non-zero checks before setting the variables which can't be changed later.
Yaodao : Recommend adding the check to check whether the `round.tokenAddress` has been set.
yekong : Add a `require` statement to check that the provided addresses are not zero addresses. This helps ensure that the contract behaves as expected and prevents potential issues related to using the zero address.
yekong : Add a check to ensure the round exists before changing the whitelist root. Also, consider adding a check to ensure the round has not started before allowing the whitelist root to be changed. For example:

```
modifier roundExists(uint32 _roundId) {
    require(_roundId <= lastCreatedRoundId, "Round does not exist");
    _;
}

function changeWhitelistRoot(uint32 _roundId, bytes32 newRoot) external onlyOwner roundExists(_roundId) {
    require(rounds[_roundId].startTime > block.timestamp, "Cannot change whitelist after round has started");
    rounds[_roundId].whitelistRoot = newRoot;
    emit WhitelistRootChanged(_roundId, newRoot);
}
```

yekong : It is recommended to add checks to the cancelRound function to ensure that the round exists and has not already been cancelled. Also, consider handling the case where a round is cancelled after participants have made pledges.

Consider the below fix in the cancelRound function:

```
function cancelRound(
    uint32 _roundId
) external onlyOwner {
    require(roundExists(_roundId), "Round does not exist");
    Round storage round = rounds[_roundId];
    require(!round.cancelled, "Round already cancelled");

    // Handle the case where round is cancelled after participants have made pledges
    if (round.totalUsdPledged > 0) {
        // Implement refund logic here...
    }

    pledgesNotSent(round);
    round.cancelled = true;
    emit RoundCanceled(_roundId);
}
```

yekong : It is recommended to add the necessary require

Client Response

Mitigated. disagree with zero-address checks L. 170 removed changeWhitelistRoot altogether L. 175 added check for existing round, not for tokenAddress as we dont deem it necessary (token deposit will never realistically happen before project got the pledges) L. 329 not necessary for our off-chain use case

DWL-4:Floating pragma should not be used

Category	Severity	Status	Contributor
Language Specific	Low	Fixed	parth_15

Code Reference

- code/src/PrivateRounds.sol#L2

```
2:pragma solidity ^0.8.13;
```

Description

parth_15 : Using a floating pragma ^0.8.13 statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.
The impact is that the testing may be done in different version while contracts can be compiled to different version which can cause bugs in the contract.

Recommendation

parth_15 : It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

Client Response

Fixed. not sure how impactful this really is, but this was adjusted

DWL-5:Incompatibility With Deflationary Tokens

Category	Severity	Status	Contributor
Logical	Low	Acknowledged	Yaodao

Code Reference

- code/src/PrivateRounds.sol#L217-L227
- code/src/PrivateRounds.sol#L250-L254
- code/src/PrivateRounds.sol#L296

```
217:        uint256 fees = _amount*FEE_BPS/10000;
218:
219:        round.totalUsdPledged += _amount;
220:        pledgedAmounts[_roundId][msg.sender] += _amount;
221:
222:
223:        stablecoinContract.safeTransferFrom(
224:            msg.sender,
225:            address(this),
226:            _amount+fees
227:        );

250:        uint256 totalFees = round.totalUsdPledged*FEE_BPS/10000;
251:        stablecoinContract.safeTransfer(
252:            feeReceiver,
253:            totalFees
254:        );

296:    function claimTokens(uint32 _roundId) external nonReentrant {
```

Description

Yaodao : When transferring standard ERC20 deflationary tokens, the input amount may not be equal to the received amount due to the charged transaction fee. As a result, an inconsistency in the amount will occur and the transaction may fail due to the validation checks. For example, if a user sends 100 deflationary tokens (with a 10% transaction fee) to the target contract, only 90 tokens actually arrive to the contract.

According to the codes in the function `claimTokens()`, the amount to transfer is recorded for the claimed amount. And `totalTokensReceived` uses the balanceOf(address(this)) and `totalClaims[_roundId]` recorded. Due to the balance of standard ERC20 deflationary tokens will change as other addresses transfer of tokens and the fees burn. As a result,

the amount can be claimed of each user will change over time with other users' claim and the change of `balanceOf(address(this))`.

```
    function claimTokens(uint32 _roundId) external nonReentrant {
        Round storage round = rounds[_roundId];

        uint256 tokenBalance = IERC20(round.tokenAddress).balanceOf(address(this));
        uint256 totalClaimedForRound = totalClaims[_roundId];
        uint256 totalTokensReceived = tokenBalance + totalClaimedForRound;

        require(totalTokensReceived > 0, "no tokens deposited yet");

        if (totalTokensReceived > round.totalTokensReceived){
            round.totalTokensReceived = totalTokensReceived;
        }

        uint pledgedUsdAmount = pledgedAmounts[_roundId][msg.sender];

        uint tokensToBeClaimed = ((pledgedUsdAmount *
            totalTokensReceived) / round.totalUsdPledged) - claims[_roundId][msg.sender];

        require(tokensToBeClaimed > 0, "no claim");

        claims[_roundId][msg.sender] += tokensToBeClaimed;
        totalClaims[_roundId] += tokensToBeClaimed;
        IERC20(round.tokenAddress).safeTransfer(
            msg.sender,
            tokensToBeClaimed
        );

        emit InvestorClaimedTokens(_roundId, msg.sender, tokensToBeClaimed);
    }
```

Yaodao : The fees of each user is calculated the amount of the user's pledge amount and transferred into the contract in the function `pledge()`. In the function `sendPledgesToProject()`, the total pledge amount of users will be used to calculate the total amount of fees and transferred to the `feeReceiver`.

In solidity, there is a truncation problem in the division calculation. So the total amount of fees calculated in the function `sendPledgesToProject()` may larger than the total fees transferred into the contract.

For example, the user A pledges 1005 tokens and the user B pledges 1995 tokens. The fee transferred from user A is `1005*1050/10000 = 105` and the fee transferred from user B is `1995*1050/10000 = 209`. The `round.totalUsdPledged` will be `1005+1995 = 3000`. Then the fee transferred to the `feeReceiver` will be `3000*1050/10000 = 315`, which is larger than the fee transferred from the user A and B.

If only exist one round, `3314` tokens transferred into the contract. Call the function `sendPledgesToProject()` need the balance of the contract is `3315` at least. As a result, the call will fail because the balance is not enough.

As the amount of round increases, the last round may occur this condition unless someone transfers the lack part directly into the contract.

Consider below codes

```
    function pledge(
        uint32 _roundId,
        uint256 _amount,
        uint256 _maxAmount,
        bytes32[] calldata merkleProof
    ) external nonReentrant {
        ...
        uint256 fees = _amount*FEE_BPS/10000;

        round.totalUsdPledged += _amount;
        pledgedAmounts[_roundId][msg.sender] += _amount;


        stablecoinContract.safeTransferFrom(
            msg.sender,
            address(this),
            _amount+fees
        );
        ...
    }

    function sendPledgesToProject(
        uint32 _roundId
    ) external nonReentrant onlyOwner{
        ...
        uint256 totalFees = round.totalUsdPledged*FEE_BPS/10000;
        stablecoinContract.safeTransfer(
            feeReceiver,
            totalFees
        );
        ...
    }
```

Recommendation

Yaodao : Recommend regulating the set of tokens supported and adding necessary mitigation mechanisms to keep track of accurate balances if there is a need to support deflationary tokens.
Yaodao : Recommend recording the amount of total fees for each round.

Client Response

Acknowledged. not relevant

DWL-6:Incorrect usage of Ownable constructor

Category	Severity	Status	Contributor
Logical	Low	Declined	yekong

Code Reference

- code/src/PrivateRounds.sol#L126-L132

```
126:    constructor(address _stablecoinAddress, address _owner, address _feeReceiver) Ownable() {
127:        stablecoinContract = IERC20(_stablecoinAddress);
128:
129:        feeReceiver = _feeReceiver;
130:
131:        _transferOwnership(_owner);
132:    }
```

Description

yekong : The contract constructor calls the Ownable constructor without passing an initial owner, and then calls _transferOwnership(_owner) in the next line. However, the version of Ownable being used requires an address parameter in its constructor to set the initial owner. As a result, the constructor of Ownable is not being used correctly, leading to unnecessary code complexity and potential confusion.

```
constructor(address initialOwner) {
        _transferOwnership(initialOwner);
    }
```

Recommendation

yekong : Modify the contract constructor to pass the _owner address directly to the Ownable constructor, and remove the redundant call to _transferOwnership(_owner). Here is a corrected example:

```
constructor(address _stablecoinAddress, address _owner, address _feeReceiver) Ownable(_owner) {
    stablecoinContract = IERC20(_stablecoinAddress);
    feeReceiver = _feeReceiver;
}
```

Client Response

Declined. incorrect, the constructor of Ownable is argumentless and transfers ownership to msg.sender

DWL-7:Lack of Necessary Checks in pledge Function Can Lead to Unexpected Behavior and Unnecessary Gas Consumption

Category	Severity	Status	Contributor
Logical	Low	Acknowledged	yekong

Code Reference

- code/src/PrivateRounds.sol#L186-L215

```
186:    function pledge(
187:        uint32 _roundId,
188:        uint256 _amount,
189:        uint256 _maxAmount,
190:        bytes32[] calldata merkleProof
191:    ) external nonReentrant {
192:        Round storage round = rounds[_roundId];
193:
194:        roundStarted(round);
195:        notCancelled(round);
196:        roundNotEnded(round);
197:
198:        require(
199:            round.totalUsdPledged + _amount <= round.groupAllocation,
200:            "exceeds groupAllocation"
201:        );
202:
203:        bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(msg.sender, _maxAmount))));
204:        require(MerkleProof.verify(merkleProof, round.whitelistRoot, leaf), "invalid proof");
205:
206:        if (block.timestamp <= round.cappedPeriodEndTime){
207:            require(
208:                pledgedAmounts[_roundId][msg.sender] == 0,
209:                "already pledged"
210:            );
211:            require(
212:                _amount == _maxAmount,
213:                "amount != maxAmount"
```

```
214:             );
215:         }
```

Description

yekong : The pledge function in the contract has several potential issues that could lead to unexpected behavior or unnecessary gas consumption:

- The function does not check whether _amount is greater than 0. If _amount is 0, users might perform useless transactions, wasting gas.
- The function does not check whether _amount+fees exceeds the user's balance. If the user's balance is less than _amount+fees, the safeTransferFrom function will fail.

Recommendation

yekong : Consider adding the following checks to the pledge function to prevent the above issues:

- Require _amount to be greater than 0.
- Before calling safeTransferFrom, check whether the user's balance is greater than or equal to _amount+fees.

Client Response

Acknowledged. not relevant, user error

DWL-8:Lack of repeatability check implement in the function `setTokenForRound()`

Category	Severity	Status	Contributor
Logical	Low	Acknowledged	Yaodao

Code Reference

- code/src/PrivateRounds.sol#L278-L294

```
278:    function setTokenForRound(
279:        uint32 _roundId,
280:        address _tokenAddress
281:    ) external onlyOwner{
282:        Round storage round = rounds[_roundId];
283:
284:        require(_tokenAddress != address(stablecoinContract), "invalid Address");
285:
286:        roundEnded(round);
287:        targetReached(round);
288:        notCancelled(round);
289:        require(round.pledgesSent, "pledges not sent");
290:
291:        require(round.totalTokensReceived == 0, "already received tokens");
292:        round.tokenAddress = _tokenAddress;
293:        emit TokenSetForRound(_roundId, _tokenAddress);
294:    }
```

Description

Yaodao : The function `setTokenForRound()` is used to set the token for the round. According to the comments, the new tokenAddress is need to check it is not already set in another project because the balance of `round.tokenAddress` is used to calculate the amounts to claim. If two projects with the same token exist at the same time, the amount that the user can claim will be calculated incorrectly. And if the token of a past project is the same as the new project, the user can also claim tokens again.

Recommendation

Yaodao : Recommend adding the variables to record the token used and check the new tokenAddress is not already set in another project.

```
    mapping(address => bool) public projectTokens;

    function setTokenForRound(
        uint32 _roundId,
        address _tokenAddress
    ) external onlyOwner{
        Round storage round = rounds[_roundId];

        require(!projectTokens[_tokenAddress]), "already set in another project");
        require(_tokenAddress != address(stablecoinContract), "invalid Address");
        ...
        round.tokenAddress = _tokenAddress;
        projectTokens[_tokenAddress] = true;
        ...
    }
```

Client Response

Acknowledged. same root issue as DWL-1

DWL-9:Gas Optimization

Category	Severity	Status	Contributor
Gas Optimization	Informational	Mitigated	parth_15, yekong

Code Reference

- code/src/PrivateRounds.sol#L54
- code/src/PrivateRounds.sol#L58
- code/src/PrivateRounds.sol#L62
- code/src/PrivateRounds.sol#L92-L109
- code/src/PrivateRounds.sol#L113
- code/src/PrivateRounds.sol#L114
- code/src/PrivateRounds.sol#L117

```
54:         require(block.timestamp >= round.startTime, "not started");

58:         require(block.timestamp <= round.endTime, "ended");

62:         require(block.timestamp > round.endTime, "not ended");

92:     struct Round {
93:         // target amount denominated in stablecoin, round considered successful if reached
94:         uint target;
95:         // max amount of stablecoin that can be pledged by the group
96:         uint groupAllocation;
97:         uint totalUsdPledged;
98:         uint32 startTime;
99:         uint32 cappedPeriodEndTime;
100:         uint32 endTime;
101:         bool pledgesSent;
102:         //@dev only updates after each claim, use receivedAndClaimable for a live value
103:         uint totalTokensReceived;
104:         address tokenAddress;
105:         bytes32 whitelistRoot;
106:         // address that receives the USDC and is eligible to set the project token
107:         address projectEOA;
108:         bool cancelled;
109:     }

113:     uint256 public immutable FEE_BPS = 1050;
```

```
114:    address public feeReceiver;

117:    IERC20 stablecoinContract;
```

Description

parth_15 : Instead of using error strings, the project should use Custom Errors. This would save both deployment and runtime cost.

The error strings consumes lot of gas compared to custom errors.

parth_15 : As the solidity EVM works with 32 bytes, variables less than 32 bytes should be packed inside a struct so that they can be stored in the same slot, this saves gas when writing to storage ~20000 gas.

The impact is that the unpacked slot in solidity consumes more slots in storage and can use lot of gas. Saving one slot can reduce the gas cost of SSTORE by ~20000.

parth_15 : `FEE_BPS` can be set to `constant` as it's value is known at compile time. While variables `feeReceiver` and `stablecoinContract` can be set to `immutable` as it's value is only assigned in constructor and not changed after deployment.

This can save gas because constant and immutable variables are stored in the contract's deployed bytecode and reading them doesn't incur cost of SLOAD or SSTORE(while writing).

yekong : In the provided Solidity contract, the `FEE_BPS` state variable is declared as public and immutable, and it is assigned a value of `1050` at declaration. Since its value is set at the time of declaration and never changed afterwards (including in the constructor), it could be declared as a `constant` instead of `immutable`.

Declaring `FEE_BPS` as a `constant` can provide some benefits. Firstly, it makes the code more readable by clearly signaling to other developers that this value is not just immutable, but a constant that will never change. Secondly, it could potentially provide some gas optimizations because constant values are inlined into the code at compile-time, whereas immutable values are stored in contract storage and read at runtime.

Recommendation

parth_15 : Use custom errors instead of require error strings.

parth_15 : Pack the struct by following ways.

The original struct used uses 8 storage slots:

```
struct Round {
        // target amount denominated in stablecoin, round considered successful if reached
        uint target;
        // max amount of stablecoin that can be pledged by the group
        uint groupAllocation;
        uint totalUsdPledged;
        uint32 startTime;
        uint32 cappedPeriodEndTime;
        uint32 endTime;
        bool pledgesSent;
        //@dev only updates after each claim, use receivedAndClaimable for a live value
        uint totalTokensReceived;
        address tokenAddress;
        bytes32 whitelistRoot;
        // address that receives the USDC and is eligible to set the project token
        address projectEOA;
        bool cancelled;
    }
```

Consider below optimized struct by reordering the field which uses 7 storage slots.

```
struct Round {
        // target amount denominated in stablecoin, round considered successful if reached
        uint target;
        // max amount of stablecoin that can be pledged by the group
        uint groupAllocation;
        uint totalUsdPledged;
        uint32 startTime;
        uint32 cappedPeriodEndTime;
        uint32 endTime;
        address tokenAddress;
        uint totalTokensReceived;
        bytes32 whitelistRoot;
        address projectEOA;
        bool pledgesSent;
        bool cancelled;
    }
```

parth_15 : Change the `FEE_BPS` to `constant` while `feeReceiver` and `stablecoinContract` to `immutable`.

yekong : Consider changing the declaration of `FEE_BPS` to:

```
uint256 public constant FEE_BPS = 1050;
```

Client Response

Mitigated. only changed to constant and immutable, not changed to custom errors and new struct order (acknowledge the validity of the optimization described though)

DWL-10:Unused code

Category	Severity	Status	Contributor
Code Style	Informational	Fixed	Yaodao, yekong

Code Reference

- code/src/PrivateRounds.sol#L10
- code/src/PrivateRounds.sol#L10-L11
- code/src/PrivateRounds.sol#L26
- code/src/PrivateRounds.sol#L46

```
10:error RoundNotStarted();

10:error RoundNotStarted();
11:error RoundEnded();

26:    event TokensDeposited(

46:    event TokensRescued(
```

Description

Yaodao : In the contract `PrivateRounds`, the error `RoundNotStarted` is never used.
Yaodao : In the contract `PrivateRounds`, the event `TokensDeposited` and `TokensRescued` are never used.
yekong : In the provided Solidity code, two custom error declarations RoundNotStarted and RoundEnded are present but appear to be unused in the contract.

While declaring custom errors in a contract does not inherently introduce a security vulnerability or a bug, it might be a sign of incomplete or unused code. If these errors are not used anywhere, they unnecessarily increase the contract's complexity and bytecode size, which can in turn increase gas costs for contract deployment.

Moreover, unused code can create confusion for developers maintaining the code. They might wonder why these errors were declared and where they are supposed to be used.

Recommendation

Yaodao : Recommend removing it to save gas if this error is not intended to be used.
Yaodao : Recommend removing them to save gas if these events are not intended to be used.
yekong : If these errors are not intended to be used in the contract, it is recommended to remove them to reduce the contract's complexity and size. If they are intended to be used but currently aren't, you should add the corresponding revert statements where these errors are supposed to be thrown.

Client Response

Fixed. removed unused Errors and Events

DWL-11:Variables that could be declared as immutable

Category	Severity	Status	Contributor
Code Style	Informational	Fixed	Yaodao

Code Reference

- code/src/PrivateRounds.sol#L114
- code/src/PrivateRounds.sol#L117

```
114:    address public feeReceiver;

117:    IERC20 stablecoinContract;
```

Description

Yaodao : The linked variables assigned in the constructor can be declared as immutable. Immutable state variables can be assigned during contract creation but will remain constant throughout the lifetime of a deployed contract. A big advantage of immutable variables is that reading them is significantly cheaper than reading from regular state variables since they will not be stored in storage.

Consider below codes

```
address public feeReceiver;
IERC20 stablecoinContract;
```

Recommendation

Yaodao : Recommend declaring these variables as immutable. Please note that the immutable keyword only works in Solidity version v0.6.5 and up.

Client Response

Fixed. changed as per recommendation

DWL-12:rename the claims mapping to claimed and totalClaims to totalClaimed

Category	Severity	Status	Contributor
Code Style	Informational	Fixed	parth_15

Code Reference

- code/src/PrivateRounds.sol#L122-L124

```
122:    mapping(uint256 => mapping(address => uint256)) public claims;
123:    //round => claimed amount
124:    mapping(uint256 => uint256) public totalClaims;
```

Description

parth_15 : Consider renaming the `claims` mapping to `claimed` as it denotes the number of tokens that are already `claimed`.

Recommendation

parth_15 : Rename the `claims` mapping to `claimed`.

Client Response

Fixed. renamed as recommended

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3’s prior written consent in each instance.

This report is not an “endorsement” or “disapproval” of any particular project or team. This report is not an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3’s position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.