PrimeDAO Seed Module V2

Final Audit Report

June 15, 2022

Team Omega

teamomega.eth.limo

Summary

PrimeDAO has asked Team Omega to audit the contracts that define the behavior of the Seed module contracts.

We found **no high severity issues** - these are issues that can lead to a loss of funds, and are essential to fix. We classified **five** issues as “medium” - these are issues we believe you should definitely address. In addition, **9** issues were classified as “low”, and **6** issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

All of these issues were resolved, except for 2 low-severity issues.

Severity	**Number of issues**	**Number of resolved issues**
High	0	0
Medium	5	5
Low	11	9
Info	6	6

Scope of the Audit

Code from the following repository and commit was audited:

```
https://github.com/PrimeDAO/launch-contracts/commit/be56604bd8ad607230f4f
3c907010c8db902ccd6
```

And specifically the following Solidity contract:

```
Seed.sol
```

Resolution

The issues mentioned in this report were addressed with commit:

```
https://github.com/PrimeDAO/launch-contracts/commit/c7852cc45f468d9b4d459
b14edde797ae7caec01
```

We reviewed these changes, and reported our observations below, under each issue.

Methods Used

Code Review

We manually inspected the source code to identify potential security flaws.

The contracts were compiled, deployed, and tested in a test environment.

Automatic analysis

We have used static analysis tools to detect common potential vulnerabilities. No high severity issues were identified with the automated processes. Some low severity issues, concerning mostly the variables and functions visibility, were found and we have included them below in the appropriate parts of the report.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

Severity definitions

High	Vulnerabilities that can lead to loss of assets or data manipulations.
Medium	Vulnerabilities that are essential to fix, but that do not lead to assets loss or data manipulations
Low	Issues that do not represent direct exploit, such as poor implementations, deviations from best practice, high gas costs, etc
Info	Matters of opinion

Findings

General

G1. Dependencies on deprecated version of OpenZeppelin [low] [partially resolved]

The `package.json` file contains four different OpenZeppelin dependencies:

```
"openzeppelin-solidity": "^4.5.0",
"openzeppelin-contracts-sol8": "npm:@openzeppelin/contracts@4.2.0"
"openzeppelin-contracts-sol5": "npm:@openzeppelin/contracts@2.5.1",
"@openzeppelin/contracts": "^4.5.0",
```

These are all references to the same npm package, `@openzeppelin/contracts`
Of these, `openzeppelin-solidity` is an outdated name for this package, version 2.5.1 is not used anywhere in the repository. Version 4.2.0 - the version that is used in the solidity files that are a subject of this audit - is an older, deprecated, version.
Recommendation: If possible, use a single dependency to the OpenZeppelin package. Refer to it by its current name `@openzeppelin/contracts`, and use the latest stable version (i.e. 4.5.0).
In addition, we recommended to include a specific version of the dependency, instead of a range - this will guarantee that your contracts will be compiled with the same dependencies, so that future developers will have a predictable outcome of the compilation process.
Severity: Low - although 4.2.0 does contain a number of bugs that were fixed in later versions, none of these bugs directly affect the contracts under consideration.
Resolution: This issue was partially resolved. The OpenZeppelin versions have been updated, but were not unified and some still allow a range instead of a specific version.

G2. Inconsistent license information in package.json [low] [resolved]

The `package.json` file suggests that the software is released under an ISC license:

```
"license": "ISC",
```

However, the solidity source code, and the included `LICENSE` file, suggest that the software is actually released under the GPL license.

Recommendation: Update the license information in package.json
Severity: Low
Resolution: This issue was resolved as recommended.

G3. Failing tests [low] [resolved]

Two of the tests are failing - we included the output of the tests in an appendix to this report.
Severity: Low
Recommendation: Fix the tests.
Resolution: This issue was resolved as recommended.

G4. Compiler warnings [info] [resolved]

The compiler gives multiple warnings which should be fixed or silenced. We have listed the warnings in the Appendix below
Severity: Info
Recommendation: Fix or silence the compiler warnings.
Resolution: This issue was resolved as recommended.

Seed.sol

S1. Sale could be started before the contract is funded [medium] [resolved]

In the `buy` function, lines 390 to 397, the following lines are used to set the `isFunded` flag:

```
if (!isFunded) {
    require(
        seedToken.balanceOf(address(this)) >=
            userClass.seedAmountRequired + userClass.feeAmountRequired,
        "Seed: sufficient seeds not provided"
    );
    isFunded = true;
}
```

We believe this check is meant to guarantee that the tokens that a user intends to buy at this step will indeed be available to claim later.

What is checked here is that the contract has enough seed tokens to cover all the buyers in the buyer's ContributorClass. However, the `isFunded` flag is a global state variable, and so, once funded for any single class, the check is skipped when users from other classes buy.

In an extreme case, a ContributorClass could be added with a `hardCap` of `0` (and therefore a value for `seedAmountRequired` of `0`), in which case users could successfully "buy" tokens from the contract even if it holds no seed tokens at all.

In any case, there is no guarantee for the funders of the contract that there will be enough seed tokens available to cover their bids.

The problem is more general, however. Users will be able to buy seed tokens for up to `hardCap` funding tokens, paying a different price and assigning different fees depending on the ContributorClass they buy the tokens in. Calculating in advance how many seed tokens will be sold is impossible, as it depends on user behavior - although calculating the maximum amount of seed token needed to cover all funders is possible

Severity: Medium. Funders may lose funds if there are no seed tokens in the contract to cover their buy (and there is no-one willing or able to add additional seed tokens to the contract) and they will not be able to retrieve their funding tokens from the contract.

Recommendation: Maintain state variables (such as `seedAmountRequired` and `feeAmountRequired`) that store the maximum amount of seed tokens that is required to cover each buying scenario, and update those each time `setClass` or `addClass` is called. Use these state variables to check if the contract is funded in the lines quoted above.

Resolution: This issue was resolved as recommended.

S2. Missing check for success of tokens transfer [medium] [resolved]

There are various places in the code where either seed tokens or funding tokens are transferred. These transfers are being called with the stand ERC20 signatures `transfer` and `transferFrom`

Due to an ambiguity in the ERC20 specifications, the behavior of these functions is not always consistent: in some implementations, an error is thrown if the transfer fails, in other implementations, a value of false is returned.

For example, in the `withdraw` function, the return value of the transfer call is not checked. This means that if the transfer fails, the value of `fundingWithdrawn` will be updated even if no tokens were withdrawn - and as a consequence, funds could be stuck in the contract.
Severity: Medium. Although unlikely, funding tokens could be stuck in the contract and not be available for the admin to claim
Recommendation: Use OpenZeppelin's `SafeERC20` library, which provide a `safeTransfer` and `safeTransferFrom` wrappers that always throw an error if the transfer fails, and replace all current calls of `transfer` and `transferFrom` with these wrapper functions.
Resolution: This issue was resolved as recommended.

S3. There is no guarantee that users can buy hardCap amount of tokens [medium] [resolved]

The contract maintains a number of state variables that are meant to keep track of `seedAmountRequired`, `feeAmountRequired`, and `seedRemainder` and `feeRemainder`. These values are set on initialization, and are calculated on the basis of the `_hardcap`, `_price` and `_fee` passed to the `initializer` function.
These variables are intended to represent upper limits on the amount of seed tokens that can be bought - i.e. `seedAmountRequired` is assumed to represent the maximum amount of seed tokens that can be bought, and the `seedRemainder` is a running counter that represents how many of these tokens are still up for sale.
However, Contributor classes can set custom prices and fees for specific accounts, which can be either lower or higher than the price set on initialization. This means that the values of state variables such as `seedAmountRequired` do not necessarily have any relation with the actual amount of seed tokens that are being sold (or reserved for fees) by the contract.
This fact has some undesirable consequences. For example, the `buy` function will subtract the amount of seeds that are needed from the "remainder" variables, and so will fail after `seedRemainder` have been sold:

```
        seedRemainder -= seedAmount;
        feeRemainder -= feeAmount;
```

As the initial value of `seedRemainder` is calculated on the basis of the initial price rather than the actual price at which tokens are being sold, this limit can be reached at any time during the sale - also before the hard cap, or even soft cap, is reached.
Severity: Medium - funders of the contract may find their orders not being filled.
Recommendation: This issue is related to S1, and we recommend in that issue to maintain a state variable that keeps proper track of the amount of seed tokens that are needed to satisfy all buyers in all Contributor Classes. In addition, we recommend that instead of keeping track of the `seedRemainder` you keep track of the amount of seeds that are distributed for sales and fees.

Resolution: This issue was resolved. Now the contract funding requirement is calculated as to fulfill for the least expensive class and the highest fee, so that there will be enough seed tokens for all classes to buy up to the hard cap of the contract and pay out all the fees.

S4. An attacker can diminish the amount of tokens available in a class [medium] [resolved]

The function `retrieveFundingTokens` allows for a funder to retract her bid and retrieve her funding tokens. However, the variable `classFundingCollected`, that keeps track of the total amount of funding collected in this specific contributor class, is not being updated.
This omission allows for an attacker that is whitelisted for a class, to buy and subsequently retrieve up to `softCap` amount of tokens in that class. After the attack, other users can now only buy `classCap - softCap` amount of tokens in this class.
Severity: Medium. Although no user directly loses funds in this attack, the attack may have off-chain financial consequences, for example if the initiator of the token sale has contractual obligations to an investor that was promised she could buy a certain amount of tokens at a certain price, and is not able to do so.
Recommendation: Add the line `classFundingCollected =- fundingAmount` to the `retrieveFundingTokens` function.
Resolution: This issue was resolved as recommended.

S5. When maximum is reached, vestingStartTime is set to the current time for only a single class [low] [resolved]

On line 427, in the `buy` function, if `fundingCollected >= hardCap` (i.e. when the hard cap is reached), the value of `vestingStartTime` for the class that the funder is buying in is set to the current time:

```
        classes[funders[msg.sender].class].classVestingStartTime =
block.timestamp;
```

This choice seems somewhat arbitrary, as the `classVestingStartTime` for the other contribution classes remain unchanged (and are set at some later date, after the `endTime`).
Severity: Low
Recommendation: Make the vesting start time a global instead of a class specific variable, or remove the line to have predictable but distinct vesting times per class.
Resolution: This issue was resolved. The vesting start time is now updated for all classes.

S6. Unused state variables [low] [resolved]

The following state variables are never read:

```
uint256 public vestingStartTime
uint256 public vestingDuration
uint256 public price
uint256 public seedClaimed
uint256 public feeClaimed
uint256 public fee
uint256 public totalFunderCount
bool public isWhitelistBatchInvoked
```

If there is no use case for making these values available for other contracts to read, they can be safely removed. If the information is needed off-chain, it is cheaper and often more effective to use events.
Severity: Low. There are some additional gas costs on deployment, and some gas costs for updating the values of these variables.
Recommendation: Remove the variable and use only the class specific variables instead, to save some gas but also for clarity
Resolution: This issue was resolved. Most variables were removed or put to use.

S7. Unused variables in ContributorClass [low] [resolved]

The struct `ContributorClass` stores a number of values that are not read, or for which it is cheaper to calculate their value on the basis of the other values in the struct:

`uint256 seedAmountRequired` (can be calculated on the basis of `price` and `classCap`)
`uint256 feeAmountRequired` (can be calculated on the basis of `seedAmountRequired` and `classFee`)

Severity: Low - some gas can be saved when adding and setting classes
Recommendation: remove these storage variables and calculate the values on the fly when they are needed.
Resolution: This issue was resolved as recommended.

S8. Counter of total funders could be manipulated upwards [low] [resolved]

In case no vesting is set (due to the require in line 405), it is possible to call the `buy` function with a funding amount of 0. This will not do anything, except increasing the `totalFunderCount` variable by 1 in each call, making its value higher than the real number of funders.
Severity: Low
Recommendation: Only increment the counter if the buy amount is greater than 0 . Or follow our recommendation in S6 and move the counter, and use an event instead.
Resolution: This issue was resolved. The `buy` function now requires the amount to be greater than 0.

S9. Add class functionality code is duplicated across multiple functions [low] [resolved]

The logic to add a class and verify its values is duplicated across the `initialize`, `addClass`, and `addClassBatch` functions.
This makes the code longer and costlier gas-wise, but also makes mistakes and inconsistencies more likely. For example, the `initialize` function requires that the `vestingDuration` will be greater than or equal to the `vestingCliff`, but the `addClass`, `addClassBatch`, and `changeClass` functions do not require that.
Severity: Low
Recommendation: Remove code duplications by moving duplicated code into a function used internally by the others instead of repeating the code. Also ensure that the value validations are consistent across the class setting functions.
Resolution: This issue was resolved as recommended.

S10. Unnecessary check that seedAmount >= vestingDuration in buy function [low] [resolved]

The `buy` function checks that the amount of seedTokens that a user buys does not exceed the `vestingDuration`:

```
        require(
             seedAmount >= vestingDuration,
             "Seed: amountVestedPerSecond > 0"
         );
```

This requirement does not check the correct values: the vesting duration for this particular buy order is not determined by the `vestingDuration` state variable, but by the `vestingDuration` of the contributor class the funder belongs to.
Secondly, even with the correct values, the requirement is not necessary and can be safely removed: if the "amount vested per second" is rounded down, the funder will still be able to retrieve all of the remaining seed tokens at the end of the vesting period - as can be checked by inspecting the `calculateClaim` function.
Severity: Low - some gas can be saved and the current check is simply incorrect.
Recommendation: Remove this check, or at least fix it to use the class data instead of the global variables.
Resolution: This issue was resolved as recommended. The check was removed from the code.

S11. Remove redundant getter of a public variable [info] [resolved]

The function `allFeeClaimed` is simply a getter for the public variable `feeClaimed`, which already has an auto generated getter due to it being public.
Severity: Info
Recommendation: Remove the `allFeeClaimed` function.
Resolution: This issue was resolved as recommended.

S12. Global array classes is missing visibility declaration [info] [resolved]

The `classes` array is missing a visibility declaration, and the code defines a getter function `getClass`. It would be better to declare it as public to reduce code and improve readability; this allows you to also remove the `getClass` function, as you can use the automatically generated getter function instead.
Severity: Info
Recommendation: Remove the `getClass` function and declare `classes` as public.
Resolution: This issue was resolved as recommended.

S13. Withdraw the whole funding token balance when withdrawing sale proceeds [info] [resolved]

The function `withdraw` calculates the amount of funding to withdraw from the contract and sends it to the sale's admin. While this is not wrong, it is more straightforward to call the funding token `balanceOf` function, and transfer the whole balance of the sale contract, which will be cheaper, and useful in case funding tokens were accidentally sent directly to the contract.
Severity: Info

Recommendation: In the `withdraw` function, send the entire seed contract balance of the funding token instead of calculating the amount to be sent.
Resolution: This issue was resolved as recommended.

S14. Mark functions external where possible [info] [resolved]

Functions that are not used within the contract itself should be declared `external` instead of `public`. This includes the feeClaimedForFunder function as well as the `allFeeClaimed` and `getClass` though those we have recommended to remove in favor of the compiler-generated getters.
Severity: Info
Recommendation: Mark functions that are not used within the contract itself as `external` instead of `public`.
Resolution: This issue was resolved as recommended.

S15. Use memory instead of reading multiple times from storage [info] [partially resolved]

In some cases a storage variable is being read multiple times or while having its value already loaded in memory. It is more gas-efficient to load storage variables to memory than read them multiple times, or read from memory instead of storage when possible. For example, see line 402 read of the `classFee`, line 744 where funder could be loaded to memory, and the function `claim` where the `classes[currentId]` is read from storage multiple times.
Severity: Info
Recommendation: Make sure to load storage variables to memory when it is being read multiple times, and to read from memory instead of storage while possible.
Resolution: This issue was partially resolved. In some places, but not all, variables are now loaded to and read from memory instead of reading them from storage multiple times.

S16. Fee is set to highest classFee, minimalPrice is set to the lowest classPrice, but is never reset [low] [not resolved]

These lines in `calculateSeedAndFee` set the fee to the highest classFee.

```
if(_classFee > fee){
       fee = _classFee;
 }
 if(_price < minimalPrice){
        [...]
```

```
            minimalPrice = _price;
    }
```

But it does not reset it if a class with the highest price (or classFee) is changed to a lower value.
The effect is that the `minimalPrice` can be underestimated, and the maximal fee overestimated.
Severity: Low. Although no funds will be lost, the funder of the contract will have to provide more Seed tokens than could ever be sold.
Recommendation: Add logic in `calculateSeedAndFee` to update these variables.
Resolution: This issue was not resolved.

S17. The calculation of seedAmountRequired can underestimate the amount of seed tokens required [medium] [resolved]

The following lines recalculate the amount of seeed tokens required to cover the sale when the price of a specific class changes.

```
     if(_price < minimalPrice){
            seedAmountRequired = (seedAmountRequired * (hardCap - _classCap) /
hardCap)
      + (_classCap * PRECISION) / _price;
            minimalPrice = _price;
```

This does not work. Consider the following scenario:

- hardCap is 1000
- class 0, classCap 1000, price is 10
    - seedAmountRequired = 1000/10 = 100
- addClass 1, classCap 200, price 5
    - seedAmountrequired is now 100*(1000-200)/1000 + (200/5) = 80 + 40 = 120
- addClass 2, classCap 200, price = 4
    - seedAmountrequired is now 120*(1000-200)/1000 + (200/4) = 96 + 50 = 146

Now we do the sale.

1. We sell all of class 2 (price is 4, so we sell 50 seed tokens for 200 funding tokens),
2. We sell all of class 1 (price is 5, so we sell 40 seed tokens for 200 funding tokens)
3. We then sell out all of class 0 until we reach the hardCap, i.e. for 600 funding tokens the buyers should get 60 seed tokens.

That adds up to 150, and so we run out of tokens.

Severity: Medium - there will not be enough seedTokens to cover the entire sale, and buyers can lose money.
Recommendation: Fix the calculation to ensure the minimum of seed tokens in the contract is enough to cover the maximum possible purchases of seed tokens.
Resolution: The issue was resolved. The current calculation overestimates the amount of seed tokens required, but the funder can always recover them after the sale has ended.

S18. Seed tokens funding required to start the contract might be higher than the highest amount that can possibly be sold [low] [not resolved]

In lines 410 to 419 (of the fixes commit), there is a check meant to disallow starting to buy tokens before the contract has sufficient seed tokens to give the buyers after the sale, as well as sufficient fee to send to the beneficiary address. This check is based on the `seedAmountRequired` and `feeAmountRequired` variables, which in turn are calculated based on the class with the lowest price, and the class with the highest fee, respectively. This however could lead to a requirement to fund the contract with more seed tokens than can actually be sold, since the cap of the classes with lowest price and highest fee may be lower than the total cap of the sale.
In short, the contract might require its funder to overfund it with seed tokens for the sale to be possible to start.
Severity: Low. No funds are at risk.
Recommendation: Fix the calculation to correctly track the maximum amount seed tokens that can possibly be sold, and the maximum fee that could be needed. This would require taking into consideration each class cap, price and fee, as well as recalculating properly in case a class cap, price, or fee is being modified.
Resolution: This issue was not resolved. The developers indicated that in any case, the surplus seed tokens can be recovered by the funder after the sale has ended.

Appendix

Test output

```
 147 passing (10s)
  2 failing

  1) Contract: Seed
       » creator is avatar
         # claim
           » claim after vesting duration
             it cannot claim before currentVestingStartTime:
```

```
     Uncaught Error: invalid BigNumber value (argument="value",
value="713e24c43730000", code=INVALID_ARGUMENT, version=bignumber/5.5.0)
      at Logger.makeError (node_modules/@ethersproject/logger/src.ts/index.ts:225:28)
      at Logger.throwError (node_modules/@ethersproject/logger/src.ts/index.ts:237:20)
      at Logger.throwArgumentError
(node_modules/@ethersproject/logger/src.ts/index.ts:241:21)
      at Function.BigNumber.from
(node_modules/@ethersproject/bignumber/src.ts/bignumber.ts:291:23)
      at NumberCoder.encode
(node_modules/@ethersproject/abi/src.ts/coders/number.ts:25:27)
      at
/Users/jelle/omega/primedao/primedao-seed-audit/node_modules/@ethersproject/abi/src.ts
/coders/array.ts:71:19
      at Array.forEach (<anonymous>)
      at pack (node_modules/@ethersproject/abi/src.ts/coders/array.ts:54:12)
      at TupleCoder.encode
(node_modules/@ethersproject/abi/src.ts/coders/tuple.ts:54:20)
      at AbiCoder.encode (node_modules/@ethersproject/abi/src.ts/abi-coder.ts:112:15)
      at Interface._encodeParams
(node_modules/@ethersproject/abi/src.ts/interface.ts:325:31)
      at Interface.encodeFunctionData
(node_modules/@ethersproject/abi/src.ts/interface.ts:380:18)
      at
/Users/jelle/omega/primedao/primedao-seed-audit/node_modules/@ethersproject/contracts/
src.ts/index.ts:225:37
      at step (node_modules/@ethersproject/contracts/lib/index.js:48:23)
      at Object.next (node_modules/@ethersproject/contracts/lib/index.js:29:53)
      at fulfilled (node_modules/@ethersproject/contracts/lib/index.js:20:58)

  2) Contract: Seed
       » creator is avatar
         # claim
           » claim after vesting duration
             it cannot claim before currentVestingStartTime:
     Error: done() called multiple times in test <Contract: Seed » creator is avatar #
claim » claim after vesting duration it cannot claim before currentVestingStartTime>
of file /Users/jelle/omega/primedao/primedao-seed-audit/test/seed.js
```

Compiler Warnings

```
Warning: This declaration shadows an existing declaration.
   --> contracts/seed/Seed.sol:756:9:
    |
756 |         uint256 price,
    |         ^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> contracts/seed/Seed.sol:35:5:
   |
35 |     uint256 public price; // price of a SeedToken, expressed in fundingTokens,
with precision of 10**18
   |     ^^^^^^^^^^^^^^^^^^^^
```

```
Warning: This declaration shadows an existing declaration.
   --> contracts/seed/Seed.sol:757:9:
    |
757 |         uint256 vestingDuration,
    |         ^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> contracts/seed/Seed.sol:40:5:
   |
40 |     uint32 public vestingDuration;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^


Warning: This declaration shadows an existing declaration.
   --> contracts/seed/Seed.sol:761:9:
    |
761 |         uint256 seedAmountRequired,
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> contracts/seed/Seed.sol:33:5:
   |
33 |     uint256 public seedAmountRequired; // Amount of seed required for
distribution
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^


Warning: This declaration shadows an existing declaration.
   --> contracts/seed/Seed.sol:762:9:
    |
762 |         uint256 feeAmountRequired)
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> contracts/seed/Seed.sol:34:5:
   |
34 |     uint256 public feeAmountRequired; // Amount of seed required for fee
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```