byterocket

Smart Contract Audit Report

Prime Deals

Token Swaps

2nd of May 2022

Contents

1. Preface **3**

2. Manual Code Review **4**

2.1 Severity Categories 4

2.2 Summary 5

2.3 Findings 6

3. Protocol/Logic Review **17**

4. Summary **18**

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held understanding of the auditor's knowledge of security patterns as they relate to the client's contract(s), assuming that blockchain technologies, in particular, will continue to undergo frequent and ongoing development and therefore introduce unknown technical risks and flaws. The scope of the audit presented here is limited to the issues identified in the preliminary section and discussed in more detail in subsequent sections. The audit report does not address or provide opinions on any security aspects of the Solidity compiler, the tools used in the development of the contracts or the blockchain technologies themselves, or any issues not specifically addressed in this audit report.

The audit report makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, the legal framework for the business model, or any other statements about the suitability of the contracts for a particular purpose, or their bug-free status.

To the full extent permissible by applicable law, the auditors disclaim all warranties, express or implied. The information in this report is provided "as is" without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. The auditors hereby disclaim, and each client or user of this audit report hereby waives, releases and holds all auditors harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.

1. Preface

The developers of **Prime Deals** contracted byterocket to conduct an audit of their smart contracts. PrimeDAO is building "*a novel interface for DAO to DAO interactions, such as token swaps, co-liquidity provision, and joint venture formation.*" Their first release is limited to token swaps between multiple DAOs, while other modules will follow.

The team of byterocket reviewed and audited the above smart contracts in the course of this audit. We started on the 18th of March and finished on the 2nd of May 2022.

The audit included the following services:

- *Manual Multi-Pass Code Review*
- *Protocol/Logic Analysis*
- *Automated Code Review*
- *Formal Report*

byterocket gained access to the code via a private GitHub repository, which will be published here close to launch. We based the audit on the main branch's state on May 2nd, 2022 (commit hash *2eb2fb582c42952a1877d13854e67d640f6fe02d*).

2. Manual Code Review

We conducted a manual multi-pass code review of the smart contracts mentioned in section (1). Four different people went through the smart contract independently and compared their results in multiple concluding discussions.

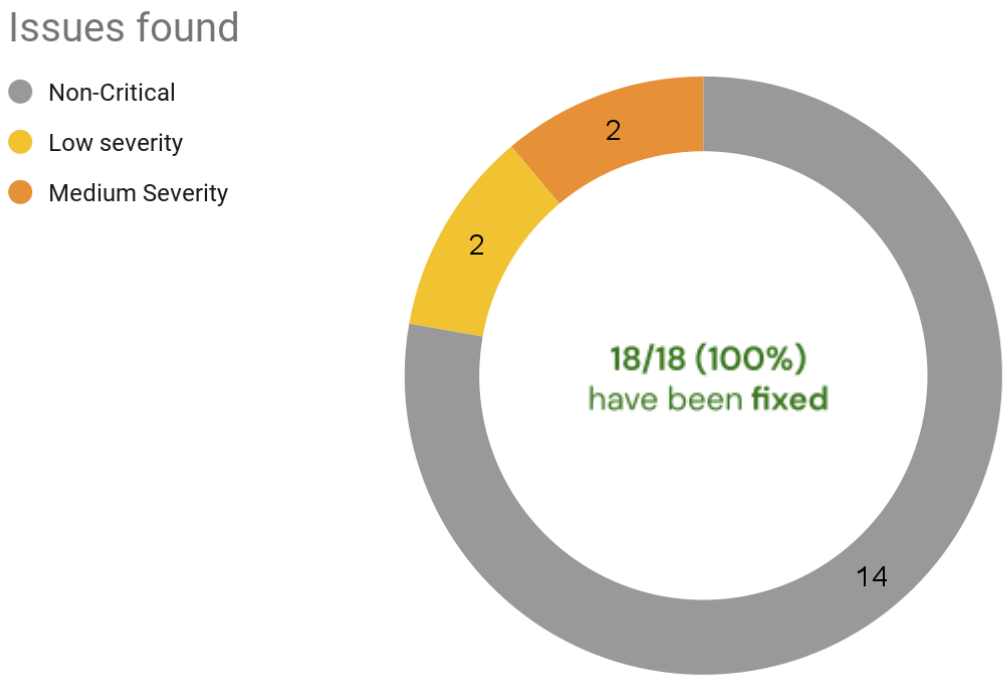
The manual review and analysis were additionally supported by multiple automated reviewing tools, like Slither, GasGauge, Manticore, and different fuzzing tools.

2.1 Severity Categories

We are categorizing our findings into four different levels of severity:

Non-Critical	Does not impose immediate risk but is relevant to security best practices. Includes issues with - Code style and clarity - Versioning - Off-chain monitoring
Low Severity	Imposes relatively small risks or could impose risks in the long-term but without assets being at risk in the current implementation. Includes issues with - State handling - Functions being incorrect as to specification - Faulty documentation or in-code comments
Medium Severity	Imposes risks on the function or availability of the protocol or imposes financial risk by leaking value from the protocol if external requirements are met.
High Severity	Imposes catastrophic risk for users and/or the protocol. Includes issues that could result in - Assets being stolen/lost/compromised - Contracts being rendered useless - Contracts being gained control of

2.2 Summary



On the code level, we **found 18 bugs or flaws, with 18 of them being fixed** in a subsequent update**.** Prior to this, there have been 14 findings that were non-critical, 2 of low severity, and 2 of medium severity.

The contracts are written according to the latest standard used within the Ethereum community and the Solidity community's best practices. There are multiple gas optimizations in the contracts, making them suitable for an Ethereum Mainnet deployment. The naming of variables is very logical and understandable, which results in the contract being useful to understand. The code is well documented. The developers provided us with a test suite as well as deployment scripts. The test suite featured a very high coverage as well as properly written tests.

2.3 Findings

[FIXED] ~~[MEDIUM]~~ M.1 - Wrong recipient for ETH transfers

Location: DaoDepositManager.sol - Line 565

Description:
In the regular transfer function within the DaoDepositManager, there is a general transfer function that encapsulates ETH as well as ERC20 transfers into one single function. For ETH-based transfers, the recipient of the transfer has been set to `msg.sender` instead of `_to`, resulting in the transfers going to the wrong address.

These are the affected lines of code:

```
(bool sent, ) = msg.sender.call{value: _amount}("");
```

Recommendation:
Consider changing the recipient from `msg.sender` to `_to`.

Update on the 31st of March:
The developers have updated the code section to transfer ETH to the correct recipient.

[FIXED] ~~[MEDIUM]~~ M.2 - CheckExecutability function can be wrong

Location: TokenSwapModule.sol - Line 216 - 247

Description:
The `checkExecutability` function in the `TokenSwapModule` could return `true` even though the conditions are not met. The function checks whether the available deal balance is sufficient compared to the `pathFrom` variable. However, if the same token is present more than one time the check is not working as intended, considering the following example:

- Available deal balance for token `A = 500`
- There are two elements in the pathFrom array for token `A`, one being `400` and the other being `200`
- The checks succeed because `500 > 400` and `500 > 200`, but the real amount of A tokens needed is `400 + 200 = 600` with `!(600 < 500)`.

Recommendation:
Depending on the product decision, consider adding a check in the `createAction` functions that no token occurs two times or add complexity in the `checkExecutability` function by computing the sum of the same `pathFrom` element for the same token and

then check `getAvailableDealBalance(..., token-that-occurs-more-than-once) < sum`.

Update on the 31st of March:
The developers have added a check to the `createSwap` function in order to ensure that no token is added to the path twice.

[FIXED] ~~[LOW]~~ L.1 - Array length not reduced correctly

Location: DaoDepositManager.sol - Line 412 - 425

Description:
In the claiming logic of vested tokens, there is a `for`-loop that works with a cached array length. Since the length of this array is cached, removing items from the original array does not change the cached length, hence it has to be reduced by 1 for each removed entry. This is currently not done, which causes the vesting to fail in certain instances.

These are the affected lines of code:

```
uint256 arrLen = vestedTokenAddresses.length;
for (uint256 i; i < arrLen; ++i) {
  if (vestedTokenAddresses[i] == token) {
    // if it's not the last element
    // move the last to the current slot
    if (i != arrLen - 1) {
      vestedTokenAddresses[i] = vestedTokenAddresses[arrLen - 1];
    }
    // remove the last entry
    vestedTokenAddresses.pop();
  }
}
```

Recommendation:
Consider adding a `--arrLen` after the removal of the array item.

Update on the 31st of March:
The proposed fix has been implemented accordingly.

[FIXED] ~~[LOW]~~ L.2 - Add reentrancy prevention to module

Location: TokenSwapModule.sol - Line 253 - 284

Description:
As the protocol allows its users to work with arbitrary tokens, the team has no control over their corresponding implementation. If there are malicious tokens, they could

potentially execute reentrancy attacks on the `executeSwap` function, since the status of the deal is only set to `done` in the end.

These are the affected lines of code:

```
function executeSwap(uint32 _dealId) external activeStatus(_dealId) {
  TokenSwap storage ts = tokenSwaps[_dealId];
  require(checkExecutability(_dealId), "Module: swap not executable");

  [...]

  ts.status = Status.DONE;
  ts.executionDate = uint32(block.timestamp);
  emit TokenSwapExecuted(address(this), _dealId);
}
```

Recommendation:
Consider moving the status and `executionDate` writes higher up in the function to prevent reentrancy attacks from happening.

Update on the 2nd of May:
The developers have moved the writes to happen directly after the `require` statement.

[FIXED] ~~[NO SEVERITY]~~ NC.1 - Inconsistent schema for IDs

Location: Throughout the project

Description:
Within the contracts, there are multiple ways on how IDs are handled. While some contracts start with IDs at 1, some start at 0, which makes not only working with the contracts from the front-end confusing but also poses a risk for integrations on the smart contract level.

Recommendation:
Consider changing the way that IDs are handled to a common standard.

Update on the 31st of March:
The developers have unified the IDs to all start at 1.

[FIXED] ~~[NO SEVERITY]~~ NC.2 - State variables should be immutable if possible

Location: Throughout the project

Description:
There are several state variables that could be `immutable`, as they should never be changed (e.g. `MAX_FEE` or `BPS`). Variables that should not be changed should be set to `immutable` as this prevents accidental changes later on as well as saves some gas.

Recommendation:
Consider setting any state variables that are fixed to `immutable`.

Update on the 31st of March:
The developers have set all of the suitable state variables to `immutable`.

[FIXED] ~~[NO SEVERITY]~~ NC.3 - Unoptimized for-loops

Location: Throughout the project

Description:
Throughout the project, there are several `for`-loops that are not optimized. For example this `for`-loop

```
for (uint256 i = 0; i < 10; i++) {
  ...
}
```

could be optimized to

```
for (uint256 i; i < 10; ++i) {
  ...
}
```

Recommendation:
Consider optimizing all of the `for`-loops to save some gas.

Update on the 31st of March:
The developers have optimized every `for`-loop in the project according to our recommendation.

[FIXED] ~~[NO SEVERITY]~~ NC.4 - Documentation on fee is wrong

Location: ModuleBaseWithFee.sol - Line 15 - 16

Description:
The documentation for the fee is partly wrong as it states 10.000 BPS equaling to 1% while it equals to 100% (100 BPS = 1%). As the rest of the implementation suggests, BPS is implemented correctly everywhere else, with the exception of this comment.

These are the affected lines of code:

```
// Fee in basis points (1% = 10000)
uint32 public feeInBasisPoints;
```

Recommendation:
Consider updating the documentation to be correct.

Update on the 31st of March:
The developers have updated the documentation to properly reflect the usage of BPS and percent.

[FIXED] ~~[NO SEVERITY]~~ NC.5 - Consider a max fee of less than 100%

Location: ModuleBaseWithFee.sol - Line 59 - 66

Description:
With the current implementation, it would technically be possible to set the fee to 100%. While it is up to the discretion of the governors to set a reasonable fee, we always encourage setting a reasonable upper limit directly in the code.

These are the affected lines of code:

```
require(msg.sender == dealManager.owner(), "Fee: not authorized");
feeInBasisPoints = _feeInBasisPoints;
emit FeeChanged(feeInBasisPoints, _feeInBasisPoints);
```

Recommendation:
Consider adding a maximum limit to the fee and enforcing it in the `setFee` function.

Update on the 31st of March:
The developers have implemented an immutable `MAX_FEE` variable that limits the fee to 20%. The maximum value is enforced correctly in the `setFee` function.

[FIXED] ~~[NO SEVERITY]~~ NC.6 - Only emit events if a variable changes

Location: Throughout the project

Description:
Currently, events for state variables are always emitted when its' corresponding function has been called, even if the value hasn't changed. It's usually the best practice to only emit an event if the state variable has been changed.

This is one of the affected lines of code:

```
feeInBasisPoints = _feeInBasisPoints;
emit FeeChanged(feeInBasisPoints, _feeInBasisPoints);
```

Recommendation:
Considering adding an `if`-clause prior to changing the value to emit the event properly.

Update on the 31st of March:
The developers have added an `if`-clause according to our recommendation throughout the project.

[FIXED] ~~[NO SEVERITY]~~ NC.7 - Unchecked layers in arrays

Location: ModuleBase.sol - Line 56 - 73

Description:
The `_path` variable submitted to the `_pullTokensIntoModule` function is a two-dimensional array that has to conform to certain requirements. While the inner layer is checked for each element of the array, the outer layer is not checked.

These are the affected lines of code:

```
amountsIn = new uint256[](_tokens.length);
require(_path.length == _tokens.length, "Module: length mismatch");
for (uint256 i; i < _tokens.length; ++i) {
  uint256[] memory tokenPath = _path[i];
  require(tokenPath.length == _daos.length, "Module: length mismatch");
  for (uint256 j; j < tokenPath.length; ++j) {
  uint256 daoAmount = tokenPath[j];
  if (daoAmount > 0) {
   [...]
  }
}
```

Recommendation:
Consider also checking the outer layer to equal to the amount of tokens in the deal.

Update on the 31st of March:
The developers have added a `require` statement to check the outer layer of the array.

[FIXED] ~~[NO SEVERITY]~~ NC.8 - Deadline is not checked in createSwap

Location: TokenSwapModule.sol - Line 105 - 130

Description:
In the `createSwap` function of the `TokenSwapModule`, there are multiple checks to ensure that the inputs conform to the requirements. However, it is not verified whether the supplied `deadline` variable is in the future.

These are the affected lines of code:

```
require(_daos.length >= 2, "Module: at least 2 daos required");
require(_tokens.length != 0, "Module: at least 1 token required");

// Check outer arrays
uint256 pathFromLen = _pathFrom.length;
require(_tokens.length == pathFromLen && pathFromLen == _pathTo.length,
        "Module: invalid outer array lengths");

// Check inner arrays
uint256 daosLen = _daos.length;
for (uint256 i; i < pathFromLen; ++i) {
  require(_pathFrom[i].length == daosLen && _pathTo[i].length >> 2 ==
          daosLen, "Module: invalid inner array lengths");
}
```

Recommendation:
Consider adding a `require` statement that verifies whether the `deadline` argument is in the future.

Update on the 31st of March:
The developers have added a `require` statement to ensure that the `deadline` is in the future.

[FIXED] ~~[NO SEVERITY]~~ NC.9 - Remove unnecessary addressIsModule function

Location: DealManager.sol - Line 115 - 117

Description:
The `addressIsModule` function is unnecessary because the `isModule` mapping is public. This means, that the compiler adds the following function automatically to the contract:

```
function isModule(address who) external view returns (address)
```

These are the affected lines of code:

```
function addressIsModule(address _address) public view returns (bool) {
  return isModule[_address];
}
```

Recommendation:
Consider using the `isModule` function instead of writing your own one.

Update on the 2nd of May:
The developers have removed the unnecessary function and reverted to using the `isModule` function.

[FIXED] ~~[NO SEVERITY]~~ NC.10 - Remove unnecessary getVestedBalance function

Location: DaoDepositManager.sol - Line 789 - 791

Description:
The `getVestedBalance` function is unnecessary because the `vestedBalances` mapping is public. This means, that the compiler adds the following function automatically to the contract:

```
function getVestedBalance(address token) external view returns (uint256)
```

These are the affected lines of code:

```
function getVestedBalance(address _token) external view returns
         (uint256) {
  return vestedBalances[_token];
}
```

Recommendation:
Consider using the `getVestedBalance` function instead of writing your own one to save some gas during deployment.

Update on the 2nd of May:
The developers have removed the unnecessary function and reverted to using the `vestedBalances` function.

[FIXED] ~~[NO SEVERITY]~~ NC.11 - Use underscores for number literals

Location: ModuleBaseWithFee.sol

Description:
There are multiple occasions where certain numbers have been hardcoded, either in variables or in the code itself. Large numbers can become hard to read.

```
uint256 large = 509125811;
```

could also be written as

```
uint256 large = 509_125_811;
```

Recommendation:
Consider using underscores for number literals to improve its readability.

Update on the 2nd of May:
The developers have updated all of the number literals to make use of underscores.

[FIXED] ~~[NO SEVERITY]~~ NC.12 - Enforce multiple deposit limit for ETH

Location: DaoDepositManager.sol - Line 152 - 159

Description:
The `multipleDeposits` function can handle only one ETH deposit, as properly documented. It would be possible to check when there is more than one ETH deposit and raise a proper error if more than one ETH deposit is included to improve UX.

These are the affected lines of code:

```
require(_tokens.length == _amounts.length, "DaoDepositManager: Error
        102");
for (uint256 i; i < _tokens.length; ++i) {
  deposit(_module, _dealId, _tokens[i], _amounts[i]);
}
```

Recommendation:
Consider enforcing that only one ETH deposit can be done with a `multipleDeposits` call due to the limits of `msg.value`.

Update on the 2nd of May:
The developers will implement a check to ensure that only one of the deposits is in Ether.

[ACKNOWLEDGED] ~~[NO SEVERITY]~~ NC.13 - Rebasing and FeeOnTransfer Tokens are not supported

Location: Throughout the project

Description:
There are tokens that are rebasing and tokens that take a fee on every transfer. Keep in mind, that these tokens are not supported in the current implementation.

Recommendation:
Consider either implementing support for these types of tokens or properly documenting this behavior. With most of these tokens working with wrapper tokens to work with external protocols, we do not see a big issue here.

Update on the 2nd of May:
The developers have acknowledged the fact that rebasing and FeeOnTransfer tokens are not supported. They have properly documented this and will openly communicate this to their community.

[FIXED] ~~[NO SEVERITY]~~ NC.14 - Invalid update flow

Location: DaoDepositManager.sol - Line 100 - 105

Description:
The `setDealManagerImplementation` function can only be called by the `DealManager` contract. However, the `DealManager` contract does not have a function to call this function, which renders this update flow invalid.

These are the affected lines of code:

```
function setDealManagerImplementation(address _newDaoDepositManager)
         external onlyDealManager {
  dealManager = IDealManager(_newDaoDepositManager);
}
```

Recommendation:
Consider adding an `onlyOwner` function to the `DealManager` that allows the governours to call the `setDealManagerImplementation` function.

Update on the 2nd of May:
The developers have added a corresponding function to the `DealManager`.

3. Protocol/Logic Review

Part of our audits are also analyses of the protocol and its logic. The byterocket team went through the implementation and documentation of the implemented protocol.

The repository itself contained tests and documentation. We found the provided unit tests that are coming with the repository execute without any issues and cover the most important parts of the protocol. With the deployment to our internal testnet, we found the protocol to be working as intended after the findings of section (2.3) have been fixed.

According to our analysis, the protocol and logic are working as intended.

We were **not able to discover any additional problems** in the protocol implemented in the smart contract.

4. Summary

During our code review (*which was done manually and automated*), we **found 18 bugs or flaws, with 18 of them being fixed** in a subsequent update**.** Prior to this, there have been 6 findings that were non-critical, 2 of low severity, and 2 of medium severity. Our automated systems and review tools did **not find any additional ones**.

The protocol review and analysis did neither uncover any game-theoretical nature problems nor any other functions prone to abuse.

In general, there are some improvements that can be made, but we are **very happy** with the overall quality of the code and its documentation. The developers have been very responsive and were able to answer any questions that we had.