OpenZeppelin

Beta Finance Audit

OPENZEPPELIN SECURITY | AUGUST 19, 2021

Security Audits

The Beta Finance team asked us to review and audit some of their protocol smart contracts. We looked at the code and now publish our results. To download a PDF of this audit, click here.

Scope

We audited commit `9e73eaf40e5118125d6925b29e4fb9ad8bf8e113` of the `beta-finance/beta` repository. This repository is private, and the links in this report will only work for the Beta Finance team at the time of writing. In scope were the following contracts:

```
BToken.sol
BetaBank.sol
BetaInterestModelV1.sol
WETHGateway.sol
BTokenDeployer.sol
BetaConfig.sol
BetaOracleUniswapV2.sol
BetaToken.sol
```

Notably, the 5 `BetaRunner` contracts were all out of scope for this audit.

Update: *After our audit, the Beta finance team fixed the issues to the extent outlined in this report, and then created a new public github repository. Commit*

System Overview

Beta Finance is a permissionless money market for lending, borrowing, and short selling crypto assets. They aim to both tolerate and reduce the risk of the high price volatility found in crypto markets.

At the center of the protocol, the `BetaBank` contract manages all money markets and positions, handling the transferal of funds both in and out of the protocol. Each individual token's market is then managed by a `BToken` contract, which tracks that particular token's availability and debt at any given time. When a user lends tokens to the protocol, they are minted `BToken`s to represent the underlying tokens they have claim over.

Positions in the `BetaBank` are not directly alterable by users, unless they are "whitelisted owners" – instead a set of contracts called the `BetaRunner`s interact with the bank on behalf of users. These Runners integrate with various decentralized exchanges to enable users to short, repay, and flash repay their positions. As stated above, the Runners were not in scope for this audit.

Privileged Roles

At deployment, the protocol sets addresses for a `governor` for the various contracts in the system. The `governor` has the power to initiate a transfer of the `governor` role to a new address. Initially, the `governor` will be the Beta Finance team, but they have expressed their intent to transfer this role to the community in the future.

A number of actions in the protocol are callable only by the `governor`. These include:

- Pausing and unpausing the `BetaBank`.
    - When paused, the bank will not allow users to alter their positions by adding and removing collateral, or borrowing and repaying debt. Additionally liquidations will not be allowed. `BToken`s linked to a paused `BetaBank` will not allow minting nor burning.
- Change the oracles used to calculate prices throughout the system.
- Update the interest model used to calculate interest on debt.

Items of Note

While auditing the protocol we noted a couple of areas of concern that are not included in the report as issues due to being out of scope. So as to be transparent about our findings we include them here.

Contract Testing

While auditing protocols we do not audit the test suite that tests the smart contracts in scope. However, while auditing this commit of the Beta Finance contracts, we noticed a few things of concern in the tests. Namely:

- One of the tests, namely `test_betabank_altruistic_liquidate`, does not pass at all. ***Update:*** *Fixed in commit `a49ecc0`.*
- At least one of the tests does not test what it is supposed to. This test should be using the second account (`a[1]`). ***Update:*** *Fixed in PR#80.*
- Due to errors encountered with the test suite's tooling, it was not possible to execute `brownie coverage` in the repository and get a full coverage report. As a result, the OpenZeppelin team were unable to verify the level of test coverage the repository has.

We view having a good suite of tests as critically important when developing smart contracts that will be holding user funds. We recommend the Beta Finance team improve their test suite going forward.

Use of `tx.origin`

The use of `tx.origin` in smart contracts is known to pose security risks, and has been the root cause of a number of exploits in the space. It is, however, possible to use `tx.origin` safely.

The `BetaBank` makes extensive use of the `allowActionFor` function for access control related to position management. One of the conditions where access is permitted is if `_owner == tx.origin && runnerWhitelists[_sender])`. The first part of this conditional on its own would be prone to exploitation, however here it is coupled with a whitelist controlled by the

Since we have not audited the Beta Runners, it is hard for us to evaluate the robustness of this access control mechanism, and to be certain about the security of this usage of `tx.origin`. Before entering production, we strongly recommend that any contracts that are to be whitelisted in the `runnerWhitelists` undergo audits as well.

Findings

Here we present our findings.

Critical severity

None.

High severity

[H01] Incorrect Uniswap price use discourages liquidations

When the protocol considers that a position does not have enough collateral, decided by the asset's `liquidationLTV`, anyone can call the `BetaBank` `liquidate` function to liquidate the position in question. The caller can pay off up to 50% of the position's debt, and in return gets paid from the collateral of the position.

The process of liquidation follows the following steps:

1. The `BetaBank` contract checks that the loan is in a state that requires liquidation.
2. The repayment of debt is made.
3. The value of the debt tokens is converted to collateral tokens, and an additional incentive bonus is added.
4. The position's state is updated to have less debt (as some has been paid off) and less collateral (as some is being given to the caller).
5. The caller is sent their collateral reward.

However, in step 3 of this process, the `BetaOracleUniswapV2` contract incorrectly calculates the conversion between debt and collateral tokens. This means that the caller can get severely have been gathered from UniswapV2. Namely, on line 154 of BetaOracleUniswapV2, the `convert` function uses the reciprocal of the `WETH/toToken` price to convert from WETH to `toToken`, instead of gathering the correct `toToken/WETH` price.

To protect against price manipulation, UniswapV2 uses Time-Weighted Average Prices (TWAPs). These TWAPs enable contracts to fetch an average price over a given time interval, instead of using the asset spot price at the current instant. This means that prices are significantly harder to manipulate, which is a great safety feature for protocols using these prices. However, while the spot price of `token0/token1` can be inverted to get the spot price of `token1/token0`, the same is not true of TWAPs: the TWAP of `token0/token1` *cannot be inverted* to get the TWAP of `token1/token0`. The more volatile the price of `token0/token1`, the more extreme this result is.

For example, assume we have tokens `WETH` and `collateral`, where the price of `WETH/collateralToken` at time 1 is 100, and the price at time 5 is 700. With some simplification of scaling, the cumulative prices tracked in UniswapV2 would be:

```
cumulativeWETH: (1*100)+(4*700) = 2900
cumulativeCollateral: (1*1/100)+(4*1/700) = 11/700
```

The average prices over the window from 0 to 5 are therefore calculated as:

```
averagePriceWETH: 2900/5 = 580
averagePriceCollateral: 11/700/5 = 0.00314 (approx.)
```

If instead of using the collateral price of `0.00314`, the inverted WETH price is used, this is a price of `1/580 = 0.00172`. This is approximately 54% of the correct price. This leads to the caller of the `liquidate` function being paid just 54% of the value that they were supposed to be paid.

Consider updating the `BetaOracleUniswapV2` to track the UniswapV2 TWAP for assets in both directions instead of merely tracking the price of `WETH/token` for every token. This will

> The prices used in Beta will be TWAP of ETH-based prices. This is to ensure consistency when performing calculations that may involve multiplication/division of TWAP prices.
>
> Here are some example calculations on how much difference can our oracle price be from the suggested approach. Using an inverse is equivalent to using harmonic means v.s. arithmetic means.
>
> In practice, the difference is very small. For example, 20% price fluctuation will result in AM of (1 + 1.2) / 2 = 1.1 and HM or 2 / (1/1 + 1/1.2) = 1.09 → only ~0.8% diff.
>
> In an extreme case of 100% price swing, AM = (1 + 2) / 2 = 1.5 and HM = 2 / (1/1 + 1/2) = 1.33 → only ~13% difference. The number is relatively small relative to the actual extreme price swing.
>
> Additionally, Beta does not rely on TWAP alone for pricing. The protocol intends to integrate with a more robust oracle source (e.g. Alpha Aggregator Oracle) for certain markets to ensure a reliable price source for these markets.

Medium severity

None.

Low severity

[L01] Cannot `liquidate` positions with only a single `debtShare`

In `BetaBank`, the `liquidate` function requires that the amount of debt to be repaid “Must not exceed half debt.” That requirement is enforced on line 372 with the conditional `debtShare <= pos.debtShare / 2`.

If there is only a single `debtShare` in the position, then the right-hand side of the inequality will always be zero. The result is that is it impossible to liquidate a position with only a single `debtShare`.

__Update:__ Fixed in PR#66.

[L02] Duplicated code

Duplicating code leaves the codebase prone to the introduction of errors. Errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical.

In the Beta codebase the logic, variables, and events that make a contract "governable" are duplicated in the `BetaConfig`, `BetaBank` and `BetaOracleUniswapV2` contracts.

This includes:

- Event: `SetGovernor`
- Event: `SetPendingGovernor`
- Event: `AcceptGovernor`
- Variable: address public `governor`
- Variable: address public `pendingGovernor`
- Function: `setPendingGovernor`
- Function: `acceptGovernor`

Relatedly, the implementation in `BetaOracleUniswapV2` is inconsistent with the other implementations of "governable" in the codebase. There, the check for the `governor` is made inline as opposed to using an `onlyGov` modifier.

To increase readability and maintainability of the codebase and to make the implementations of the same patterns more consistent, consider consolidating duplicated code into a single contract and inheriting from it whenever the duplicated functionality is required.

__Update:__ Not fixed. The Beta team stated that they wish to maintain the current design in order to retain the ability to emit different error messages in the various contracts.

[L03] Errors and omissions in events throughout codebase

Many of the event definitions are lacking indexed parameters. Some examples include:

- The events in [illegible].
- The events in `BetaOracleUniswapV2`

Some events only include the new value of a state variable being set. It is considered best practice to emit both the old and the new values. Some examples include:

- The governor-related events in `BetaOracleUniswapV2`, `BetaBank`, and `BetaConfig`.
- The SetX events in `BetaBank`.

Finally, the `AcceptGovernor` event can be removed in all contracts, replacing it with the `SetGovernor` event.

Consider making the above changes to the codebase to avoid hindering the task of off-chain services searching and filtering for events.

Update: *Partially fixed in PR#67.*

[L04] `initialize` can be frontrun

The `BetaBank` is designed with upgradeability in mind, and uses an `initialize` function rather than a constructor to facilitate future upgrades. This is a common upgradeability pattern, but since the `initialize` function is public, anyone can call it with arbitrary or malicious values. If the contract is not initialized in the same transaction as it is constructed, then a legitimate actor could be frontrun by a malicious actor when calling `initialize`.

Consider initializing contracts within the same transaction as their construction to be a priority in the design of the upgrade scheme and deployment mechanisms for this project. Alternatively, consider limiting who can call the `initialize` function.

Update: *Acknowledged. The Beta team state:*

> Keep as is. During deployment, we use proxy deploy pattern.

[L05] Interest rate tier deviates from whitepaper

However, the corresponding code in `BetaInterestModelV1` sets the range where the interest rate remains stable as .7 <= u < .8. Note that the range there is *exclusive* of `.8`.

To reduce confusion and clarify intentions, consider bringing the implementation and the whitepaper into agreement.

Update: *Not fixed. The Beta team state:*

> At 0.8, the rate won't get updated anyways, so it's the same.

[L06] Lack of documentation

There are several instances throughout the codebase where NatSpec is missing or incomplete. Some examples of this are:

- Functions in WETHGateway are missing NatSpec `@return`s.
- `WETHGateway`'s constructor is missing NatSpec comments.
- `BToken`'s constructor is missing NatSpec comments.
- `BetaOracleUniswapV2` is missing NatSpec `@parameter`s and `@return`s for most of its functions.

Additionally, some of the existing NatSpec contains erroneous information. Examples of this include:

- The `BToken` `borrow` function incorrectly labels `debtShare` as a `@param` but it is a `@return`.
- The `BetaBank` `selflessLiquidate` function's `@param _amount` incorrectly states that it "Must not exceed half debt."

Additionally, inline comments generally are severely lacking throughout the codebase:

- Global variables, such as those in `BToken`, should be explained.
- Constants, such as those in `BToken`, should be explained.
- Modifiers, such as those in `BetaBank`, should be explained.

comments explaining the logic they implement.

To increase the overall quality and readability of the codebase, consider updating and augmenting the current NatSpec and inline documentation so that all functions, modifiers, variables, and constants are well documented.

Update: *Fixed in PR#68.*

[L07] Lack of input validation

While several of the functions in the codebase that take external input do verify those inputs, there are also several instances of constructors and functions that lack sufficient input validation. This is especially true of functions that take input from “trusted” parties such as the `governor`. Even a trusted party may make an error while inputting values, and many clients pass along zero values by default for empty fields.

Instances of functions and constructors lacking input validation include:

- The `setConfig`, `setOracle` and `setInterestModel` functions in `BetaBank` all accept zero address inputs.
- The `BToken` and `BetaOracleUniswapV2` constructors both accept zero address inputs. The variables they accept zero addresses for are `immutable`, so a new deployment would be necessary to correct any mistakes.
- The `initialize` function of `BetaBank` accepts zero address inputs.
- The `pause` and `unpause` functions in `BetaBank` do not check the pause state first, potentially wasting SSTOREs.

Consider implementing `require` statements where appropriate to validate all user-controlled input.

Update: *Fixed in PR#69.*

[L08] `liquidate` rounding leads to incorrect `debtValue`

In the `BetaBank` `liquidate` function, the `_amount` of debt specified is repaid by calling `BToken` `repay`, which then returns the number of `debtShares` that were paid off. The amount.

To increase the accuracy of the liquidation function, consider having `BToken` `repay` return the value of the debt paid off, or alternatively consider looking up the amount repaid with 2 calls to `totalLoan`.

Update: *Not fixed. The Beta team state:*

> This is the intended behavior.

[L09] Magic constants

Literal values expressed in scientific notation are used throughout the codebase. For example:

- On line 441 of `BetaBank`, where `1e18` is returned by the `_fetchPositionLTV` function. The literal value is used elsewhere in the same function.
- The `BetaConfig` contract uses the literal value `1e18` extensively to place upper bounds on rates and loan-to-value ratios.
- Throughout `BetaInterestModelV1` hard-coded numeric values are used.

These hard-coded numeric values often lack any sort of explicit explanatory inline documentation, making the code difficult to read and understand.

To improve readability and overall clarity, consider defining a `constant` variable for every magic value used, and giving each a clear and self-explanatory name. Alternatively, consider adding detailed inline comments explaining how the literal values were calculated or why they were chosen.

Update: *Not fixed.*

[L10] Using old Solidity version

Throughout the codebase, Solidity 0.8.3 is used. However, at the time of writing there is one known bug in version 0.8.3. Consider upgrading the codebase to use the latest version of Solidity, 0.8.6.

Update: *Fixed in PR#83.*

example, this happens on line 103 in `BetaOracleUniswapV2`. Although divisions by zero will result in reversions, they will not have error messages, making failed transaction more difficult to debug by users.

Consider actively preventing divisions by zero with appropriate `require` statements that have informative and user-friendly error messages.

Update: *Fixed in PR#69. `minTwapTime` can no longer be `0`.*

[L12] Using `tx.origin` excludes smart contract wallets

The `allowActionFor` function in `BetaBank` takes two arguments, a position owner and a message sender. It returns true if the provided sender is allowed to act on behalf of the provided owner.

The `onlyOwner` modifier in the `BetaBank` contract is simply a wrapper around `allowActionFor` that always provides `msg.sender` as the sender argument. This modifier is used on most position management functions in `BetaBank`.

The result is that to fully interact with `BetaBank`, a caller must either be:
1. The owner of a position and also whitelisted in the `ownerWhitelists` mapping or
2. The initiator of a transaction where the caller to `BetaBank` is in the `runnerWhitelists` (ostensibly this whitelist is reserved for Beta runners).

The runners are outside the scope of this audit, but it is apparent from the second condition alone that smart contract wallets, unable to satisfy the requirement that `_owner == tx.origin`, will not be able to interact with `BetaBank` via runners. This means that such wallets will be unable to use the Beta Finance protocol as it is intended to be used.

If this is desirable or expected behavior, consider explicitly documenting this restriction alongside the `allowActionFor` function. Otherwise, consider modifying conditions so that they do not exclude smart contract wallet users from interacting with `BetaBank` via runners.

Update: *Not fixed. The Beta team state:*

[N01] Undocumented implicit approval requirements

Throughout the codebase, the contracts implicitly assume that they have been granted an appropriate allowance before calling `transferFrom` or `safeTransferFrom`. For instance:

- Within the `burn` function in `WETHGateway`
- Within the `put` function of `BetaBank`
- Within the `mint` function of `BToken`

In favor of explicitness and to improve the overall clarity of the codebase, consider documenting all approval requirements in the corresponding functions' comments.

Update: *Not fixed.*

[N02] Incomplete interface

The interface `IBetaConfig` is missing several public functions from the corresponding `BetaConfig` contract. Missing functions include `cFactors`, `rLevels`, `rConfigs`, and `getRiskLevel` among others.

To help clarify intentions and make integrating with the project less error prone, consider including all `public` and `external` functions in the appropriate interfaces throughout the codebase.

Update: *Partially fixed in PR#70.*

[N03] Misleading conditional

On line 82 of the `BToken` contract there is a conditional that tests whether the `uint timePast` is less than or equal to zero (`<= 0`). As `timePast` is a `uint`, it cannot ever be less than zero.

To increase overall code clarity and readability, consider limiting conditionals so that they only compare against values that are possible.

Update: *Fixed in PR#71.*

future changes. There are some names in the code that make it confusing or hard to understand.

Consider the following suggestions:

In the `BetaBank` contract

- The `runnerWhitelists` and `ownerWhitelists` variables and the `setRunnerWhitelists` and `setOwnerWhitelists` functions are unnecessarily pluralized.
- The `allowActionFor` function should be renamed to `isPermittedCaller`.
- The `lastCumu variable` and the `currentPrice0Cumu` and `currentPrice1Cumu` functions should be renamed to include the complete `Cumulative` word.
- The `exts` variable should be renamed to `externalOracles`.
- The `take` function should be renamed to `takeCollateral`.
- The `put` function should be renamed to `putCollateral`.
- The `borrow` function should be renamed to `borrowUnderlying`.
- The `repay` function should be renamed to `repayUnderlying`.
- The `checkPID` modifier should be renamed to `checkPIDExists`.
- The `onlyOwner` modifier should be renamed to `isPermittedByOwner`.

In the `BToken` contract

- The `totalAvailable` variable should be renamed to `totalLoanable`.

***Update:** Partially fixed in PR#72. Some of the above changes were made.*

[N05] Non-standard implementation of ERC20 metadata

While ERC20 metadata – including `decimals`, `name`, and `symbol` – are all optional parts of the ERC20 standard, they are often relied upon by off-chain services to facilitate user interactions with ERC20 tokens. These off-chain services, ranging from centralized exchanges to wallet clients, often use metadata to present user balances and token information in a user-friendly manner.

cannot be relied upon.

While we did not identify a security risk stemming from this behavior in the codebase itself, it may cause unnecessary issues with projects wishing to integrate with `BTokens`. Since non-stable metadata is abnormal, it could potentially deter other projects from integrating with `BTokens`. In the worst case, some popular off-chain service could make assumptions about stable metadata that could prove to be problematic and confusing for `BToken` users.

Consider making `BToken` metadata stable to simplify ecosystem integrations in the future.

Update: *Not fixed.*

[N06] Inconsistent use of SafeERC20

Contracts throughout the codebase that handle ERC20 token transfers and approvals utilize OpenZeppelin's `SafeERC20` and its methods to perform the function calls safely. However there are a handful of occurrences where this is not the case, and instead the transfers and approvals are performed as calls directly to the token.

While each of these calls is to a currently "known token" and not to arbitrary ERC20s, future token versions and changes could cause these direct function calls to be unsafe.

Consider using `SafeERC20` throughout the entire codebase to ensure all token calls are handled safely.

Update: *Fixed in PR#73.*

[N07] Not using latest stable version of OpenZeppelin Contracts

An old version of OpenZeppelin Contracts, version 4.0.0, is being used throughout the codebase. Consider using the latest stable release of the library, which is version 4.2.0 at the time of writing.

Update: *Fixed in PR#74.*

[N08] Redundant `require`

revert rather than return.

This `getCollFactor` function is only ever called by the `BetaBank` (here and here), and whenever it is called it is accompanied by an additional `require` statement that checks that the return value is greater than zero. The result being that the return value of `getCollFactor` is always subjected to redundant `require` statements.

Consider removing the redundancy to save gas, and simplify the codebase.

Update: *Not fixed. The Beta team states:*

> Config may get updated and this extra check will serve as a safety measure for bad config contract implementation.

[N09] Reimplementing OpenZeppelin Contracts library

The `lock` modifier in `BetaBank` implements custom logic to prevent reentrancy of security-critical functions. Although this does not pose a security risk, consider always inheriting functionality from the secure, community-vetted, OpenZeppelin Contracts library.

In particular, consider inheriting from the OpenZeppelin `ReentrancyGuard` contract, which is already used in the `BToken` contract. This will help reduce the code's attack surface.

Additionally, the `IERC20Metadata` interface in `BToken` reimplements the OpenZeppelin `IERC20Metadata` interface. Consider instead utilizing this community-vetted implementation.

Update: *Not fixed.*

[N10] Rounding calculation can overflow

In lines 172 and 116 of `BToken`, the result of an integer division is rounded up. The addition contained in this calculation might overflow if the underlying token operates with extremely large amounts.

Although it is unlikely to encounter a token that can lead to this issue, given the permissionless nature of the system it is not possible to prevent a user from using such a token. Consider using

__Update:__ Fixed in PR#76.

[N11] Typos

The codebase contains the following typos:

- "adjust the rate" should be "adjusts the rate".
- "Not constructor" should be "No constructor".
- "borrow" should be "borrowed".
- "is is" should be "is".
- `timePast` should be `timePassed` on line 81 of `BToken` and line 37 of `BetaInterestModelV1`.
- "decimals" should be "decimal" in lines 20 and 69 of `BToken`.
- "Must not exceed half debt" should be removed.

Consider correcting these typos to improve code readability.

__Update:__ Fixed in PR#77.

[N12] Unused named return

The `deploy` function in `BTokenDeployer` names its return value as `bToken`. However ultimately uses an explicit return, not the named return.

Consider removing unnecessary named returns to clarify the intentions and improve the readability of the codebase.

__Update:__ Fixed in PR#78.

[N13] Unused variable

On line 402 of the `BetaBank` contract, the variable `underlying` is declared, but it is never used.

Consider removing unused variables to reduce the size of the codebase and improve overall code clarity.

Across the codebase, there are hundreds of instances of `uint`, as opposed to `uint256`. The lack of explicitness can make the codebase less readable, for instance `type(uint).max` is not as clear to reason about as the more verbose `type(uint256).max`.

In favor of explicitness, consider replacing all instances of `uint` with `uint256`.

Update: *Not fixed.*

Conclusions

0 critical and 1 high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.

To download a PDF of this audit, click here.