VinuFinance Security Companion Document (EVM Version, v1.1)

Samuele Marro

January 5, 2024

1 Introduction

This document provides relevant security-related information for the `BasePool` and `Controller` contracts, and explains the security rationale behind several design decisions. It should be read in conjunction with ChainSecurity's audit of the original contract[1] and the security companion document of the Vite version [2] This document is not meant to replace the documentation of the code, but rather provide insight into potential challenges and considerations when using, maintaining and upgrading the contracts.

2 Deployment Process

The recommended steps to deploy the contracts are:

1. Setup a multisig for the veto holder (if necessary)
2. Deploy the `Controller` contract
3. Check that the `Controller` contract has been deployed successfully by querying a method (e.g. `Controller.vetoHolder`)
4. Deposit a non-trivial amount of vote tokens (ideally enough to have a majority share for at least the first few days, assuming regular usage of the pools) by calling `Controller.depositVoteToken`
5. Deposit all the remaining vote token supply (except for the amount that will be airdropped to users and developers) by calling `Controller.depositRewardSupply`
6. For each pool:
   (a) Deploy the `BasePool` contract
      - Keep in mind that `BasePool` reverts if `loanTenor < 86400` or `creatorFee > 3%` (see Section 3.3)

   (b) Check that the `BasePool` contract has been deployed successfully by querying a method (e.g. `BasePool.getPoolInfo`)
   (c) Create a proposal to whitelist `BasePool` by calling `Controller.createProposal`
   (d) Vote on the proposal by calling `Controller.vote`
   (e) Set the veto holder approval by calling `Controller.setVetoHolderApproval`
   (f) Check that the pool has been correctly whitelisted by querying `Controller.poolWhitelisted`
   (g) Deposit a small amount of liquidity in `BasePool` by calling `BasePool.addLiquidity`
   (h) Force a reward update by calling `BasePool.forceRewardUpdate`
   (i) Check that the reward has been correctly disbursed by querying `Controller.rewardBalance`
7. Airdrop vote tokens to users and developers
8. (Optionally) setup `MultiClaim` and `EmergencyWithdrawal`

[1] https://chainsecurity.com/wp-content/uploads/2022/10/ChainSecurity_MYSO_Finance_Core_Protocol_V1_audit_221018.pdf
[2] https://github.com/Vita-Inu/VinuFinance-Vite/blob/master/Security%20Companion%20Document.pdf

3 Security-Related Design Decisions

This section lists several design decisions that are related to contract security. The intention is to allow future maintainers and/or developers to understand the rationale behind some design decisions in order to avoid accidentally creating vulnerabilities. For the sake of brevity, design decisions that were already present in Myso's original implementation are not discussed.

3.1 Rejecting a Higher Repayment than Expected

Myso's original contract allows a user to successfully repay a loan with an amount that is arbitrarily higher than the expected amount. This decision provides no benefit for the users and can instead cause users to accidentally lose funds. For this reason, my implementation of `BasePool` requires users to repay with *exactly* the expected amount.

3.2 Loans on Behalf of Someone Else

The original `BasePool` implementation allows users to take loans on behalf of someone else, without the latter's authorization. While a bit unusual, the loan is still transferred to the official borrower, which has no obligation to repay the loan (and thus claim the collateral deposited by the original caller). Therefore, the worst thing a user can do by taking a loan on behalf of someone else is giving free money to the latter. Additionally, this approach allows a smoother integration with other DeFi contracts (e.g. by allowing a contract to implement loans as part of their financial structures). For these reasons, I did not remove this feature.

3.3 Value Checks in BasePool Constructor

The original `BasePool` constructor contains checks on the values of `loanTenor` and `creatorFee` (specifically, it reverts if `loanTenor < 86400` or `creatorFee > 0.3%`). I elected to keep these checks, although I replaced the 0.3% threshold with a 3% one. It is possible to change these thresholds by modifying `MIN_TENOR` and `MAX_FEE` in the source code.

3.4 Contract Interaction Uses Duck Typing

`Controller` has very few checks on whether the contracts it is managing are actually instances of `BasePool`: this has the positive side effect of allowing `Controller` to potentially control other types of contract, such as newer versions of `BasePool` or even completely different contracts.

3.5 Pausing and Unpausing do not Require Whitelisting

It is possible for `Controller` to pause (and unpause) non-whitelisted contracts. The reason behind this design decision is that unlike in the case of reward requests (which require making sure that the request is from a legitimate source), there are no security benefits for a pause whitelist (since there is the assumption that contracts not meant to be paused by `Controller` will simply reject pause requests). Since the whitelist process involves an additional vote and an approval from the veto holder, adding it would increase the complexity of the pause process for no security gains.

3.6 Vote Token Withdrawal is Forbidden when Voting

In `Controller`, it is not possible to withdraw vote tokens when some votes are active. This prevents users from casting a vote, withdrawing the vote tokens, depositing them in another account and casting again a vote. A minor unpleasant side effect of this approach is that if a vote passes or fails, a user needs to first manually remove its vote before being able to withdraw. The reason behind this design decision is that it prevents unbounded quota consumption (since "unlocking" all voter accounts would require an arbitrary amount of quota).

3.7 Bandwagoning Prevention

It is possible for users to deposit vote tokens right before a token snapshot is about to be taken (thus being able to claim a share of the fees despite depositing the vote tokens for a short period). To combat this phenomenon, `Controller` has a minimum time period `lockPeriod` before being able to withdraw tokens, thus limiting deposit-claim fees-withdraw cycles.

3.8 Snapshots are Token-Specific

Each snapshot tracks a single token, which means that it is possible (and frequent) for different tokens to be snapshotted at different times. There are two main reasons for this decision:

- It prevents unbounded quota consumption (since the quota usage does not depend on the number of tokens tracked by the pool)
- It allows the contract to support arbitrary revenue sources without first requiring a vote to keep track of a given token.

3.9 Complex Timestamps

A token snapshot also keeps track of the exact total balance of vote tokens in the instant the snapshot was taken. However, it might be possible for a subsequent transaction (which is still in the same block) to deposit vote tokens and claim the proceeds of the snapshot (since `block.timestamp` does not distinguish between transactions in the same block). In order to prevent this phenomenon, each `Controller` operation in the same block increases `subTimestampCounter`, which allows the contract to determine whether a transaction in the same block happened before or after the snapshot.

3.10 Veto Approval is not Boolean

Instead of being a Boolean, the approval of a whitelist proposal is stored as an address. When checking if a proposal is approved by the veto holder, `Controller` checks whether the stored approver matches the current veto holder. This behavior has two benefits:

1. Changing the veto holder automatically invalidates all previous approvals (in $O(1)$ time)
2. By setting the veto holder to the zero address, it is possible to remove the need of an approver for whitelist proposals

3.11 Account Vote is not Boolean

Instead of being a Boolean, whether an account voted for a proposal is stored as an `uint256` containing the vote token balance of the voter. This simplifies updating the total vote count when removing a vote. Note that, in case an account receives additional vote tokens, it is possible to update the vote token amount by removing the vote and casting it again.

3.12 Majority is Determined Based on Current Quorum

When determining if the total number of votes is higher than the required threshold, `Controller` uses the *current* total number of votes. This means that it is possible for a proposal to retroactively reach a quorum, e.g.:

1. Alice has 10 votes, Bob has 90 votes
2. Alice votes for a proposal. The proposal now has 10% of the votes
3. Bob withdraws his vote tokens. The proposal now has 100% of the votes.

Similarly, depositing new vote tokens can increase the vote total supply, increasing the quorum.

This phenomenon is intended behavior: `Controller` should reflect the will of the *current* vote token holders, which may or may not be the same as that of previous token holders. Note that it is possible to trigger a quorum check by removing a vote and casting it again.

Additionally, note that `Controller.createProposal` requires a `_deadline` parameter, which prevents the execution of a proposal after its intended deadline.

3.13 BasePool Allowance is Temporary

When `BasePool` deposits controller fees by calling `Controller.depositRevenue` (which transfers tokens from `BasePool` to `Controller`), the allowance is increased and then immediately decreased. While requiring more gas, this approach protects the user funds against potential manipulations of `Controller`.

4 Potential Security Concerns

This section details potential aspects of the contracts that users and maintainers need to be aware of. While none of these can be considered vulnerabilities, it is still important to keep in mind these phenomena while interacting with, deploying and upgrading the contracts.

4.1 Forged Pool Information

It is extremely important to verify, before whitelisting a pool, that the code of the pool matches the official implementation or, in case of a new version of the pool, that it has been thoroughly audited. The reason is that, when the pool calls `requestTokenDistribution`, it is not possible for `Controller` to verify the truthfulness of the parameters. As a consequence, it might be possible for a malicious pool to request an arbitrarily high reward by claiming that a user deposited a large amount of tokens in a pool for a long period of time.

Note that checks on the values (e.g. setting a maximum liquidity or a maximum duration) can be circumvented by claiming that an arbitrarily high number of users are entitled to a reward.

It is possible to limit this phenomenon by setting a threshold on how many reward tokens can be awarded in a given period of time, but any threshold low enough to prevent an economically meaningful attack would also risk not awarding funds to genuine users during periods of high activity. Additionally, the news of an exploit that allows the attacker to claim undue rewards would crash the value of the token, regardless of the actual value of the rewards. For this reason, I decided not to implement such a limit, and instead set a higher bar for pool whitelisting by explicitly requesting the veto holder's approval.

4.2 ERC20s Can Block Transfers

While the contracts are designed to be resistant against revert-based DoS attacks (e.g. a contract address reverting when accepting transfers in the native token of the chain), the ERC20s used in the contracts could theoretically block transfers for malicious purposes. Pool deployers should thus carefully check the code of the chosen ERC20s to ensure that they are standard-compliant. One could argue that the fact that there is no approval process for ERC20s used in `Controller` could put the operation of the contract at risk; however, the functions used in `Controller` are token-specific, in the sense that, if a malicious ERC20 were added to `Controller`, only the contract calls related to that specific token could be disrupted. The only exception is the vote token, which is chosen at deployment time and is assumed to be vetted by the deployers.

4.3 Block Stuffing

Compared to the Vite version, the EVM implementation of the contract is less affected by DoS attacks due to the fact that the burden of paying the quota is entirely on the caller. That said, especially in low-fee and zero-fee chains, block stuffing attacks can still be performed for malicious purposes (e.g. by DoSing the chain right before the expiration of a loan); for this reason, my recommendation is to repay a loan slightly before the expiration of a loan, in order to make a DoS attack financially unsustainable.

4.4 Liquidity uint128 Computation in Reward

In order to avoid overflows when computing rewards in `Controller`, the liquidity parameter is a `uint128` (instead of `uint256`) variable. For this reason, requesting a reward will fail if a user's liquidity cannot be expressed as a `uint128`. This can be solved by removing part of the liquidity. Note that, for a regular token with 18 decimals, a `uint128` can store a maximum value of $\sim 3 \cdot 10^{20}$ units, i.e. 300 billion billions. For tokens with less decimals, this value is even higher.

4.5 Reward Coefficient is a uint96

Again due to potential overflows in `Controller`, the reward coefficient is a `uint96` variable denominated in `BASE`, which means that it supports a fixed-point value between 10^{-18} and $\sim 8 \cdot 10^{10}$. This means that the reward coefficient can only have a value in this range (although, for essentially all tokens, this range is more than sufficient).

4.6 Dust in Controller

When claiming the revenue of a snapshot, rounding errors can cause `Controller` to return slightly less ($\sim 1/10^{14}$th) than the expected amount. This issue also affects reward computation. For this reason, it is possible for an extremely small amount of funds (i.e. "dust") to be locked in the contract. Since the order of magnitude of the dust is around 10^{-14}, I elected not to take steps to prevent the accumulation of dust, as it would further increase the complexity of `Controller`.

4.7 Timestamp uint32 Overflow

The contracts store the timestamps as a `uint32` variable. For this reason, the contracts will store timestamps correctly until February 7th, 2106 (about 83 years from now, at the time of writing). In case a further extension of operation time was needed, the recommended solution is to remove the funds and deploy new contracts that use `uint64` instead of `uint32` (which is guaranteed to work for $\sim$ 500 billion years).

4.8 (Useless) DoS

If the token holders were to vote for all contracts to be paused, it might be possible to indefinitely stop any future loans. This might allow an attacker to perform an arbitrarily long DoS by buying a large percentage of all vote tokens and then voting on its own to pause the contracts. However, this possibility can be safely ignored on account of the fact that doing so would be an extremely unwise decision, from a financial point of view: the attacker would have to make a significant purchase of tokens only to then immediately make them worthless (since no further revenue from the pools would be awarded to holders).

4.9 Nullified Security Concerns

Due to the specifics of the EVM, the following security concerns, which were present in the Vite version, have been completely nullified:

- 4.2 Vite Block Stuffing Variant
- 4.3 Contract Deployment Needs to be Checked
- 4.9 Out-of-Quota can be Mistaken as Revert
- 4.10 Default Vuilder Compiler is Unoptimized
- 4.11 Both Contracts Need Quota for Reward Distribution

4.10 BasePool Tolerates Failures in Controller

In order to prevent errors in `Controller` from blocking transactions in `BasePool` (such as removing liquidity and claiming interest), `BasePool` will tolerate and ignore reverts in `Controller`. While this improves the robustness of `BasePool`, it can also hide potential issues with `Controllller`.

4.11 Controller Provides Partial Rewards if Lacking Funds

Since only `BasePool` can request token distribution with `Controller.requestTokenDistribution` (which is called automatically by `BasePool` without any user input), in case of failure due to lack of funds `Controller.requestTokenDistribution` has two options: reverting (thus causing the entire transaction to revert) or awarding the available funds (which can be potentially zero). We decided to follow the latter approach, since guaranteeing a correct behavior of `BasePool` takes priority over providing rewards. In any case, the `Controller` deployers should ensure that the funds in `Controller` are enough to cover future rewards. Note that, even with `BasePool` tolerating failures in `Controller`, awarding partial rewards is better than reverting and thus not awarding any rewards.