veTenet Security Review

Pashov Audit Group

Conducted by: pashov

August 10th, 2023

Contents

1. About pashov 3

2. Disclaimer 3

3. Introduction 3

4. About veTenet 4

5. Risk Classification 5

5.1. Impact 5

5.2. Likelihood 5

5.3. Action required for severity levels 6

6. Security Assessment Summary 6

7. Executive Summary 7

8. Findings 10

8.1. Critical Findings 10

[C-01] Cloning a subGauge can result in an uninitialized proxy 10

[C-02] Codebase is using a vulnerable Vyper compiler version 11

[C-03] New subGauges can't be used by the protocol 11

8.2. High Findings 13

[H-01] Changing the reward rate results in non-claimable yield 13

8.3. Medium Findings 14

[M-01] Input is insufficiently validated in multiple methods 14

[M-02] Non-standard ERC20 tokens will be stuck in contracts 15

[M-03] Validator fee will be lost if recipient is not set 15

[M-04] The owner of AddressRegistry can take over the protocol 16

8.4. Low Findings 17

[L-01] A paused RewardVault shouldn't allow setting reward rate 17

[L-02] A killed TenetVesting should have withdrawable return 0 17

[L-03] Iterating over unbounded arrays can result in DoS 17
[L-04] Inconsistent salt argument when cloning subGauges 18
[L-05] No way to withdraw funds during emergency in RewardVault 18
[L-06] Implementation contract can be initialized 18
[L-07] Use a two-step governance role transfer pattern 18

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **veTenet** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About veTenet

`veTenet` is deployed on Tenet, which is an EVM-compatible Layer-1 blockchain that uses LSDs (liquid staking derivatives) as the collateral that validators in the network stake. The protocol itself empowers the holders of the native `TENET` asset with voting rights, enabling them to influence the distribution of token incentives in different gauges by on-chain voting. It borrows logic & design from Curve DAO's voting & reward mechanism. `veTenet` uses a unique per-block rewards distribution mechanism based on validator performance.

More docs

Observations

While `TENET` tokens are transferable, the `veTENET` tokens are not until their lock period (up to 2 years) ends.

Voting power of `veTENET` tokens decays over time, with a maximum boost of 2.5x for 2 years of locking.

`TENET` tokens will be minted directly to the `RewardVault` contract.

The owner of the `AddressRegistry` contract has total control over the protocol.

More unit testing needs to be done in the protocol as it is far from 100% code coverage.

Privileged Roles & Actors

- Governance - can call most methods in the codebase, for example can set the daily reward rate in `RewardVault`
- TLSD Factory - can create new `subGauge`s or set the `validatorFee` or the `validatorFeeRecipient` in `GaugeProxy`
- Address Registry owner - can set all addresses used by the protocol, including the `governance` one
- Emergency - can call `togglePause` in the `RewardVault` contract
- Vesting Beneficiary - is expected to receive vested rewards

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash **-** **9771131e6e21a4dc804e729026228f6e2092415a**

fixes review commit hash **-** **434bcc0ac716d2143dccd01eef74e16e97116cdb**

7. Executive Summary

Over the course of the security review, pashov engaged with veTenet to review veTenet. In this period of time a total of **15** issues were uncovered.

Protocol Summary

Protocol Name	veTenet
Date	August 10th, 2023

Findings Count

Severity	**Amount**
Critical	3
High	1
Medium	4
Low	7
Total Findings	**15**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Cloning a subGauge can result in an uninitialized proxy	Critical	Resolved
[C-02]	Codebase is using a vulnerable Vyper compiler version	Critical	Resolved
[C-03]	New subGauges can't be used by the protocol	Critical	Resolved
[H-01]	Changing the reward rate results in non-claimable yield	High	Resolved
[M-01]	Input is insufficiently validated in multiple methods	Medium	Resolved
[M-02]	Non-standard ERC20 tokens will be stuck in contracts	Medium	Resolved
[M-03]	Validator fee will be lost if recipient is not set	Medium	Resolved
[M-04]	The owner of AddressRegistry can take over the protocol	Medium	Resolved
[L-01]	A paused RewardVault shouldn't allow setting reward rate	Low	Resolved
[L-02]	A killed TenetVesting should have withdrawable return 0	Low	Resolved
[L-03]	Iterating over unbounded arrays can result in DoS	Low	Resolved
[L-04]	Inconsistent salt argument when cloning subGauges	Low	Resolved
[L-05]	No way to withdraw funds during emergency in RewardVault	Low	Resolved

[L-06]	Implementation contract can be initialized	Low	Resolved
[L-07]	Use a two-step governance role transfer pattern	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Cloning a `subGauge` can result in an uninitialized proxy

Severity

Impact: High, as `subGauge` initialization can be front-ran

Likelihood: High, as every new `subGauge` will be vulnerable

Description

In `GaugeFactory::createSubGauge`, the `Clones::cloneDeterministic` method is used to deploy a new `subGauge`. The problem is, in contrast to `GaugeFactory::createGaugeProxyAndSubGauges`, in `createSubGauge` the `subGauge` is not initialized after cloning, which leaves it vulnerable to a front-running attack. Example scenario:

1. Alice calls `GaugeFactory::createSubGauge` to deploy a new `subGauge`
2. The `subGauge` is added to a `GaugeProxy` by calling `GaugeProxy::addSubGauge`
3. Now Alice sends a transaction to call `initialize` on the `subGauge`
4. Bob sees Alice's transaction and front-runs it, initializing it with his own supplied arguments (`stakingToken`, `governance` etc), so he controls it

Recommendations

Initialize a newly cloned `subGauge` in `GaugeFactory::createSubGauge` in the same way that it is done in `GaugeFactory::createGaugeProxyAndSubGauges`.

[C-02] Codebase is using a vulnerable Vyper compiler version

Severity

Impact: High, as reentrancy in methods that forbid it is possible

Likelihood: High, as the bytecode will always be vulnerable when compiled with that compiler version

Description

A month ago we saw a major exploit in the Ethereum ecosystem due to a bug in the Vyper compiler, as explained here. The Vyper team had the `@nonreentrant` annotation (or a function decorator) built natively into the language itself, but in version 0.2.15 a bug was introduced, essentially resulting in no reentrancy protection even if the annotation is used on a method. As stated here, "All Vyper contracts that have been compiled with versions v0.2.15, v0.2.16, and v0.3.0 are vulnerable to the malfunctioning re-entrancy guard.".

The Vyper files in the repository (apart from `BoostDelegationV2`) all have compiler version 0.2.16, which is also configured in `hardhat.config.ts`. There are multiple methods marked with `@nonreentrant` in `GaugeController`, `LiquidityGaugeV4` and `VotingEscrow`. All of those method are not reentrancy protected.

Recommendations

Upgrade all Vyper contracts' compiler version to at least v0.3.1, where the bug was fixed.

[C-03] New `subGauge`s can't be used by the protocol

Severity

Impact: High, as important protocol functionality is not working

Likelihood: High, as the bug will happen every time

Description

The `createSubGauge` method in `GaugeFactory` creates a new `subGauge` but does not actually add the `subGauge` to the `GaugeProxy` as it is done in `createGaugeProxyAndSubGauges`. Even though the `GaugeProxy` contract has an `addSubGauge` method, it is only callable by the `gaugeFactory` or by the `TLSDFactory`. Neither contracts have an actual way to directly call the `addSubGauge` method, which means that even though you can create new `subGauge`s you can't actually add them to the `GaugeProxy` in any way.

Recommendations

Change the `createSubGauge` method to execute the same `GaugeProxy(_gaugeProxy).addSubGauge(_stakingTokens[i], subGauge);` call as in `createGaugeProxyAndSubGauges` with the given `_stakingToken` argument.

8.2. High Findings

[H-01] Changing the reward rate results in non-claimable yield

Severity

Impact: High, as accrued rewards won't be distributed to validators

Likelihood: Medium, as it requires method to be called in a specific order

Description

The `setDailyRewardRate` in `RewardVault` resets the `lastReward` storage variable to the current day. This means that now when `RewardVault::distributeRewards` is called, only the duration since the latest `setDailyRewardRate` call and the current moment will be used to calculate the accrued rewards. The problem with this approach is if that before calling `setDailyRewardRate` there were accrued but unclaimed rewards, those rewards will not be claimable anymore and the validators will lose on them.

Recommendations

Call `distributeRewards` before you update `lastReward` in `setDailyRewardRate`. This should only be done when `lastReward` has already been set.

8.3. Medium Findings

[M-01] Input is insufficiently validated in multiple methods

Severity

Impact: High, as it can result of loss of accrued yield for validators or DoS of the protocol

Likelihood: Low, as it requires configuration error by `Governance` or `TLSDFactory` or them being malicious or compromised

Description

The input validation of multiple methods throughout the codebase is insufficient:

- In `RewardDistributor::setScanPeriod` if the `_newScanPeriod` argument is too big of a number it can result in a DoS in the `_distribute` method, and if it is too small it can result in lost unclaimed yield for validators
- In `GaugeProxy::setValidatorFee` if the `_fee` argument is more than `FEE_BASE` it will result in a DoS in `distributeToken` and also if it is equal to `FEE_BASE` it will result in the whole `rewardAmount` getting sent to the `validatorFeeRecipient` address
- In `TenetVesting::initialize` the `cliff`, `vestingPeriod` and `startTime` inputs are not properly validated and can contain too big values which can result in stuck funds in the contract if `revocable == false`

Recommendations

For the `scanPeriod`, make sure to not allow too big of a value and also to be certain that validators won't be using unclaimed yield. For the `validatorFee` cap it to some sensible value, possibly 10%. For the vesting parameters make sure that the `cliff` isn't too big, same for the `vestingPeriod` and also make sure that `startTime` isn't too further away in the future or already passed.

[M-02] Non-standard ERC20 tokens will be stuck in contracts

Severity

Impact: High, as it will result in stuck funds

Likelihood: Low, as only some tokens that might not actually be used will be problematic

Description

Tokens like `USDT` do not return a `bool` on `ERC20::transfer` call, while others do not revert on a failed transfer but just return `false`. Those types of tokens are not properly supported in `VestRewardReceiver::sendTokens` and `TenetVesting::rescueFunds`, as the code in those methods uses the normal `ERC20::transfer` method without accounting for them. This can result in inability to call `rescueFunds` for `USDT` for example in `TenetVesting`, which will result in stuck funds.

Recommendations

Use the OpenZeppelin `SafeERC20` library instead and change the `transfer` calls to `safeTransfer` to support those non-standard ERC20 tokens as well.

[M-03] Validator fee will be lost if recipient is not set

Severity

Impact: High, as it will result in a loss of yield for a validator

Likelihood: Low, as it requires a specific scenario

Description

In `GaugeProxy::distributeToken` rewards are distributed to gauges based on validator activity. Also some fee is taken, called a `validatorFee`, and sent to a different recipient for each validator. The problem is that the code doesn't check if a `validatorFeeRecipient` has been set - if it hasn't been then the `fee` would be burned (sent to the zero address).

Recommendations

Before doing `IERC20Metadata(tenet).transfer(validatorFeeRecipient[validators[i]], fee);` make sure to check if `validatorFeeRecipient[validators[i]] != address(0)` to not burn the `fee` sent.

[M-04] The `owner` of `AddressRegistry` can take over the protocol

Severity

Impact: High, as funds can be stolen

Likelihood: Low, as it requires a malicious or a compromised `owner` of `AddressRegistry`

Description

Currently most contracts have critical methods only callable by the `GOVERNANCE` account. This account can be set or updated at anytime by the `owner` of the `AddressRegistry` contract, which allows him to set an address that he controls and then exploit all protected methods' functionality for his own gain, for example making him the only recipient of rewards in the protocol.

Recommendations

The `owner` of the `AddressRegistry` contract should be a `Timelock` contract that is controlled by a multi-sig or a `Governor` smart contract so that actual attacks will be done slow enough so that users will have time to possibly stop using the protocol or withdraw funds from it.

8.4. Low Findings

[L-01] A paused `RewardVault` shouldn't allow setting reward rate

The `RewardVault` contract has a `pause` mechanism, which allows the contract to do nothing when `distributeRewards` is called. The problem with this is that the `setDailyRewardRate` is still callable when the contract is paused, which shouldn't be the case. Make sure to check the `isPaused` flag in `RewardVault::setDailyRewardRate` as well and revert when it is equal to `true`.

[L-02] A killed `TenetVesting` should have `withdrawable` return 0

The `TenetVesting` contract has a `kill` mechanism, which allows a vesting schedule to be revoked. Almost all methods check for the `isKilled` flag and if it is equal to `true` they revert, apart from the `withdrawable` method. To be precise, the method should return 0 when `isKilled == true`.

[L-03] Iterating over unbounded arrays can result in DoS

The `GaugeProxy` contract has the `addSubGauge` method which pushes new elements to the `gauges` and `stakingTokens` arrays. Both of those arrays' sizes should be capped due to the fact that if too many items are added to them then iterating over the arrays can become too costly in terms of gas and if the amount of gas needed is over the block gas limit this will result in a state of DoS for the methods that do this. While there is a `removeSubGauge` method, it also iterates over the arrays, so it is suboptimal. Your best approach is to introduce `MAX_GAUGES_COUNT` and `MAX_STAKING_TOKENS_COUNT` constant values to cap the sizes of the arrays.

[L-04] Inconsistent `salt` argument when cloning `subGauge`s

In `GaugeFactory` there is cloning of `subGauge`s both in `createGaugeProxyAndSubGauges` and in `createSubGauge`, but the way the `salt` argument for the cloning is computed is different. This can result in multiple `subGauge`s created for the same `stakingToken`, which shouldn't be the case. Make sure the `salt` is computed in the same way consistently.

[L-05] No way to withdraw funds during emergency in `RewardVault`

The `RewardVault` contract has pausing functionality which is expected to be called when there is some kind of an emergency, for example a possible vulnerability discovered in the contract. You should add a ways to withdraw the funds out of the contract in such case after pausing it, otherwise they can be stolen by back-running an unpause transaction to the contract. Consider changing the pausing functionality to only be set once and if it is set to `true` to have a separate method to withdraw the funds from.

[L-06] Implementation contract can be initialized

The `GaugeProxy` contract is an implementation contract that is expected to be used through a proxy. Since implementation contracts shouldn't be used, it is a convention to disallow their initialization. Consider adding an empty constructor that calls `_disableInitializers()` in `GaugeProxy`, as well as in `TenetVesting` and `VestRewardReceiver`.

[L-07] Use a two-step `governance` role transfer pattern

The `VestFactory::setGovernance` uses a single-step access control transfer pattern. This means that if the current `governance` account calls `setGovernance` with an incorrect address, then this `governance` role will be lost forever along with all the functionality that depends on it. Follow the pattern from OpenZeppelin's Ownable2Step and implement a two-step transfer pattern for the action.