TENET - veTENET

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **August 18th, 2023 - October 2nd, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	09/21/2023
0.2	Draft Review	09/21/2023
1.0	Remediation Plan	11/23/2023
1.1	Remediation Plan Review	11/23/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Ferran Celades	Halborn	Ferran.Celades@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

VeTENET is a decentralized governance and incentives platform designed to empower TENET token holders through vote-escrowed governance rights. By locking their TENET tokens as veTENET, users acquire voting power directly proportional to their staked amount, with incentives distributed among specialized gauges that represent validators, stability pools, and various DeFi protocols. With the focus on on-chain transparency and immutability, veTENET integrates a time-decaying voting power mechanism and distributes rewards based on both stake size and validator performance. The platform operates on a robust smart contract framework with a six-week rollout plan, including rigorous testing and testnet deployment, ensuring a secure and reliable ecosystem for token holders.

TENET engaged Halborn to conduct a security assessment on their smart contracts beginning on August 18th, 2023 and ending on October 2nd, 2023 . The security assessment was scoped to the smart contracts provided in the tenet-org/veTenet-contract GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided six weeks for the engagement and assigned a full-time security engineer to assessment the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some security risks that were mostly addressed by TENET . The main ones were the following:

- Review the security of TENET to veTENET conversion and confirm non-transferability during the lock period.
- Validate the integrity of on-chain voting, including decay and power adjustment mechanisms.
- Ensure accuracy in gauge allocation logic for both staked amounts and validator performance.
- Review the Fee Distributor contract for potential vulnerabilities, especially in handling multiple types of rewards.
- Assess the security of validator data inputs used in Gauge Controller calculations.
- Confirm the GaugeProxy contract's secure handling of sub-gauge rewards distribution and validator fees.
- Review the Vault contract's access controls and timing restrictions for daily reward fetching.
- Validate that the Reward Distributor operates securely and in accordance with governance rules.
- Ensure that Gauge Factory functions are securely restricted to governance access.
- Model and evaluate system behavior under edge cases like sharp changes in validator performance or governance attacks.
- Assess risks of protocol centralization affecting security and trust.
- Verify the precision and safety of arithmetic calculations across all contracts to avoid overflows or underflows.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose
- Smart contract manual code review and walkthrough

- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE, Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker’s control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

The security assessment was scoped to the following smart contracts:

- AddressRegistry.sol
- GaugeFactory.sol
- GaugeProxy.sol
- RewardDistributor.sol
- RewardVault.sol
- TenetDepositor.sol
- TenetVesting.sol
- VestFactory.sol
- VestRewardReceiver.sol

Commit: 434bcc0ac716d2143dccd01eef74e16e97116cdb

Out-of-scope:
- third-party libraries and dependencies
- economic attacks

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	1	1	1	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) OVER-DISTRIBUTION ON KILLED GAUGES	Critical (9.4)	SOLVED - 11/23/2023
(HAL-02) POTENTIAL DUPLICATE REWARD TOKEN ENTRIES	High (7.5)	SOLVED - 11/22/2023
(HAL-03) MISSING VALIDATION IN INITIALIZE FUNCTION	Medium (6.3)	SOLVED - 11/22/2023
(HAL-04) OVERRIDING DATA FOR EXISTING VALIDATORS	Low (2.2)	FUTURE RELEASE
(HAL-05) REDUNDANT REWARD COUNT CHECK IN FUNCTIONS	Informational (0.3)	ACKNOWLEDGED
(HAL-06) MISSING ZERO CHECKS	Informational (0.2)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) OVER-DISTRIBUTION ON KILLED GAUGES - CRITICAL(9.4)

Description:

The internal _distribute function in the RewardDistributor contract is tasked with distributing rewards to gauges. Within its execution, the function checks for the lastPull timestamp, the date of the most recent reward pull, and distributes rewards for each day in the past scanPeriod days that haven't yet been rewarded. However, there is an oversight in the way rewards are distributed to gauges that have been temporarily deactivated or "killed" using the toggleGauge function.

If a gauge is temporarily killed (disabled) using the toggleGauge function, no rewards are distributed to it, even if the distribute function is called with its address. If this gauge remains disabled for n days and is then reactivated, calling distribute will retroactively distribute rewards for each day in the past scanPeriod days, including those days when the gauge was deactivated.

Consider the following sequence of events:

1. A gauge A is killed (deactivated) and remains so for n days.
2. Another active gauge B has rewards distributed every day, updating the lastPull timestamp.
3. After n days, gauge A is reactivated.
4. The distribute function is called with gauge A's address.

In this scenario, gauge A will receive rewards for the days it was deactivated. This can result in an over-distribution of rewards, thereby devaluing the token for other stakeholders and causing an unfair allocation of resources.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:H/Y:N/R:N/S:U (9.4)

Recommendation:

To ensure a more efficient and fair reward distribution mechanism, the following measures are recommended:

1. **Track Last Distribution Date per Gauge**:
   - Instead of having a global lastPull timestamp, maintain a mapping of each gauge to its last distribution date. This way, you only need to look back to the last distribution date for each gauge, rather than scanning scanPeriod days back, which can save on gas and prevent over-distribution.
2. **Reset Distribution Date for Reactivated Gauges**:
   - When a gauge is reactivated after being killed, reset its last distribution date to the current date. This prevents rewards from being retroactively distributed for the days it was inactive.
3. **Optimize Storage**:
   - The isGaugePaid mapping keeps track of which gauges have been paid for which days. This can be optimized further by pruning or cleaning up entries older than the scanPeriod, saving on storage and potentially further reducing gas costs.

By implementing these recommendations, you ensure that rewards are distributed fairly, efficiently, and in a gas-optimized manner. This can lead to increased trust and engagement from stakeholders and provide a more robust rewards mechanism.

Remediation Plan:

SOLVED: The TENET team solved the issue by disabling the ability to resume a killed gauge in the future on commit cb514a79e991c36ee2bbdaddc2ce3bc3f9ded91e.

4.2 (HAL-02) POTENTIAL DUPLICATE REWARD TOKEN ENTRIES - HIGH (7.5)

Description:

The add_reward function in the LiquidityGaugeV4.vy contract lacks validation to check if the _distributor parameter is not an empty address.

The absence of this essential verification permits unintended behavior. If _distributor is provided as an empty address, the reward_count still gets incremented. Moreover, the _reward_token is set at the incremented reward_count index, despite an empty _distributor address. This allows the same _reward_token to be added multiple times at different reward_count indexes, leading to a discrepancy in reward management and potentially disrupting the rewards' distribution mechanism.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:M/Y:M/R:N/S:U (7.5)

Recommendation:

Modify the add_reward function to include a verification step, ensuring that the _distributor parameter is provided as a non-empty address:

Listing 1

```
assert _distributor != empty(address)
```

This validation will ensure that the function will not proceed if an empty distributor address is provided, safeguarding the integrity of the reward mechanism.

Remediation Plan:

SOLVED: The TENET team solved the issue by adding an empty check on commit 4dfdbeced1f86e2718c3a87a5dbf22e9b22e11ad.

4.3 (HAL-03) MISSING VALIDATION IN INITIALIZE FUNCTION - MEDIUM (6.3)

Description:

The initialize function in the LiquidityGaugeV4.vy contract, which is responsible for initializing the contract's state variables, contains a potential vulnerability related to the handling of the _gaugeProxy parameter.

The function does not validate if the provided _gaugeProxy is not an empty address. This absence of a critical verification step permits subsequent calls to the add_reward function that can mistakenly insert the TENET token as a reward token twice: once during initialization and another time via the add_reward function. This duplicity is due to the increment of reward_count in the initialize function.

The logic inside the initialize method directly sets the TENET as the reward token at index 0. If add_reward is then called for TENET, the same token will be set at index 1, resulting in unintended behavior and possibly disrupting the rewards' distribution mechanism.

BVSS:

AO:A/AC:L/AX:M/C:N/I:H/A:N/D:H/Y:N/R:N/S:U (6.3)

Recommendation:

To address this issue, you should add a validation check to ensure that _gaugeProxy is not an empty address before proceeding with the initialization.

Amend the initialize function to include a check ensuring the _gaugeProxy address is non-empty. This verification step is essential to prevent undesired behaviors during the contract's life span:

Listing 2

```
assert _gaugeProxy != empty(address)
```

Remediation Plan:

SOLVED: The TENET team solved the issue by adding an empty check on commit eec2c5321ee94c4c4db214294890881f4d02a2f8.

4.4 (HAL-04) OVERRIDING DATA FOR EXISTING VALIDATORS - LOW (2.2)

Description:

The GaugeProxy contract contains two functions, addSubGauge and distributeToken, which retrieve a validator address for a given staking token from the ITLSDFactory contract using the getValidator method.

In both functions, the retrieved validator address, validatorOf, is utilized to either initialize or update the validatorActivity mapping without checking if this validator already exists in the mapping. The mapping is intended to track the activity of each validator in terms of signed blocks, creation time, and last claim time.

Failure to check for the existence of validatorOf in the validatorActivity mapping before updating can result in the unintentional overwriting of data if the validator is associated with multiple staking tokens. This can lead to data loss and unpredictable behavior.

BVSS:

AO:S/AC:L/AX:L/C:N/I:C/A:M/D:N/Y:N/R:N/S:U (2.2)

Recommendation:

Ensure that you check if the validator already exists in the validatorActivity mapping before attempting to update it in both the addSubGauge and distributeToken functions. If the validator is present, determine the desired behavior---whether to skip the update, merge data, or handle it differently.

Remediation Plan:

PENDING: The TENET team stated that: This logic will change totally, so not relevant anymore.

4.5 (HAL-05) REDUNDANT REWARD COUNT CHECK IN FUNCTIONS - INFORMATIONAL (0.3)

Description:

In the LiquidityGaugeV4.vy contract's deposit, withdraw and _transfer function, there's a check to determine if any rewards are active. This is done by verifying if the reward_count is different from 0. However, since the contract's initializer sets the default TENET token as a reward and the reward_count is set to 1, this check will always return true.

This means that the following logic, wrapped under the condition if is_rewards:, will always be executed.

Given this, if there's any cost (in terms of gas) associated with the above operation, or other implications of always triggering the reward check pointing logic on every deposit, it can result in inefficiencies or unintended behavior.

BVSS:

AO:A/AC:H/AX:H/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.3)

Recommendation:

Since the reward_count is initialized to 1 by default and never reduced to 0 again in the given codebase, the condition self.reward_count != 0 in the aforementioned functions will always evaluate to true. To enhance clarity and reduce redundant checks, consider removing or refactoring the condition is_rewards: bool = self.reward_count != 0 and directly implement the logic inside the condition without the check. This will make the codebase leaner and more transparent to readers and developers.

Remediation Plan:

ACKNOWLEDGED: The TENET team acknowledged this finding.

4.6 (HAL-06) MISSING ZERO CHECKS - INFORMATIONAL (0.2)

Description:

The codebase contains multiple instances where critical functions are missing zero address checks, creating vulnerabilities and potentially undesired behaviors. The affected contracts and their respective issues are as follows:

1. **AddressRegistry's setAddress Function:**
   - This function associates a string name with an address. Currently, there is no check to prevent the zero address from being stored. Failing to validate this could cascade into problems wherever the address registry is used.
2. **TenetDepositor's createLockForUser and increaseAmount Functions:**
   - Both functions accept Ether and convert it to wrapped TENET tokens. Missing zero value checks could result in unnecessary gas costs and could allow users to call these functions without meaningful interactions. This assumes that the functions of external contracts handling these operations have zero checks, which may not always be the case.
3. **GaugeFactory's Constructor:**
   - The constructor initializes critical state variables with addresses of _subGaugeImplementation and _addressRegistry. A missing zero address check for these parameters could lead the contract to a malfunctioning state, affecting all its interactions and leading to potential loss of funds or accessibility.
4. **createGaugeProxyAndSubGauges and Asset Checks:**
   - The function interacts with ITLSDFactory through the getAsset method. The address returned (_lsd) is used in multiple operations, including cloning contracts and setting mapping values.

If, for any reason, getAsset returns a zero address, it could result in unintended behaviors like overwriting existing contract data or causing deployed contracts to malfunction.

BVSS:

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:F/S:U (0.2)

Recommendation:

To mitigate these vulnerabilities, introduce validation checks as follows:

1. **For AddressRegistry's setAddress function:**

Listing 3

```
require(_addr != address(0), "Provided address is zero");
```

2. **For TenetDepositor's createLockForUser and increaseAmount functions:**

Listing 4

```
require(msg.value > 0, "No Ether sent");
```

3. **For GaugeFactory's constructor:**

Listing 5

```
require(_subGaugeImplementation != address(0), "
↳ SubGaugeImplementation is zero address"); require(address(
↳ _addressRegistry) != address(0), "AddressRegistry is zero address
↳ ");
```

4. **For createGaugeProxyAndSubGauges after each call to getAsset:**

Listing 6

```
require(_lsd != address(0), "Returned asset address is zero");
```

Remediation Plan:

ACKNOWLEDGED: The TENET team acknowledged this finding.

REVIEW NOTES

In the design documentation, the validator fee formula is described as:

$$ValidatorFee = TotalGaugeRewards/fee * 100$$

$$GaugeReward_i = (TotalGaugeRewards - ValidatorFee) * \frac{ActiveBlocks_i}{\sum_{n=1}^{totalValidators} ActiveBlocks_n}$$

The formula is invalid as the validator fee is a percentage of the total gauge reward and not the other way around:

$$ValidatorFee = TotalGaugeRewards * fee/100$$

5.1 BoostDelegationV2.vy

Forked from the curve-veBoost project and BoostV2.vy contract. It includes all functionalities but the migration from BoostV1.

- It was informed that the contract was out of scope, and no more review was performed. Although found issues will be reported anyway.

5.2 VeBoostProxy.vy

Forked from Angle Protocol under https://github.com/AngleProtocol/angle-core/blob/main/contracts/staking/veBoostProxy.vy.

- It was informed that the contract was out of scope, and no more review was performed. Although found issues will be reported anyway.

5.3 GaugeController.vy

Forked from the curve-dao-contracts project and GaugeController.vy contract.

- The contract does not have a token as a parameter on the constructor, it rather converts the voting_escrow address into a VotingEscrow and extracts the token from it.
- The initializer does also set the admin as parameter rather than the creator of the contract.
- commit_transfer_ownership does check for zero address.
- accept_transfer_ownership will verify that the sender is the future_admin and store the address of the sender as the new admin. **It is a good idea to also set the future_admin to zero, which will return some used gas**
- A new gauge_exists function is added which returns a boolean based on the gauge_type being different from zero.
- The vote_for_gauge_weights was converted to an internal function with an _ prefix and a user parameter added. The external function, named vote_for_gauge_weights, does use the msg.sender as the parameter for the user.
- A new vote_for_many_gauge_weights function is added, supporting an array of _gauge_addrs and _user_weight.
- It was informed that the contract was out of scope, and no more review was performed. Although found issues will be reported anyway.

5.4 VotingEscrow.vy

Forked from the curve-dao-contracts project and VotingEscrow.vy contract.

- Max time is modified from 4 years to 2 years.
- An initializer variable was added, the constructor does initialize the contract to prevent the proxy implementation from being initialized again.

- 2 new parameters are added to the initializer, smart_wallet_checker and tenetDepositor.
- The forked version does not allow changing the admin once it is set, both commit_transfer_ownership and apply_transfer_ownership functions are removed.
- The internal _deposit_for and public version was modified to allow depositing for another address, being the sender the provider of tokens. This means that a new parameter was added indicating the msg.sender of the actual call of the deposit, where the funds will be transferred from.
- create_lock_for was also added. This function does allow locking for a different address rather than the sender. The funds are deposited for the given address, but the tokens transferred from the sender.
- The find_timestamp_epoch view function was added. This function performs a binary search but uses the timestamp instead of the block number.
- Two new view functions, named find_block_user_epoch and find_timestamp_user_epoch were added. Those functions perform the same binary search as find_block_epoch and find_timestamp_epoch respectively but using the user_point_history instead of point_history.
- The original balanceOf does grab the last user_point_epoch and uses the slope to calculate what was the balance of the user back in time. However, this is an approximation which is not that precise. The slope is essentially a rate of change for the bias over time. Using the current epoch's slope to calculate the balance for a past timestamp would give you an approximation, assuming that the rate of change (slope) has been consistent. But if there were changes in the slope during previous epochs (e.g., due to events that changed the voting power), the approximation might not be accurate. The forked version does use the provided timestamp to fetch the closest epoch registered for that user. It then uses that as the last value, the same way the original version did. This allows the final value to be more precise in case the slope does increase do to a surge in deposits from the user.
- balanceOfAt is using the internal find_block_user_epoch to perform the same as the old implementation.

- totalSupply does use the internal find_timestamp_epoch function instead of relying on the supply_at slope calculations from the last point_history. This allows, as stated on balanceOf, better precision on the closest known value for that timestamp instead of relying on the last slope rate change.
- It was informed that the contract was out of scope, and no more review was performed. Although found issues will be reported anyway.

5.5 LiquidityGaugeV4.sol

Forked from the curve-dao-contracts project and LiquidityGaugeV4.vy contract. There exist many changes in this contract.

- An initializer variable was added, the constructor does initialize the contract to prevent the proxy implementation from being initialized again.
- The deposit function will call the _checkpoint_rewards when no value is provided, with the _only_checkpoint parameter set to true. This will cause only the _checkpoint_reward to be computed for the TENET token.
- The deposit function does now allow _claim_rewards for value equal 0. Meaning that deposit cannot be called to "claim" TENET token like you would if value was provided. This is some sort of discrepancy, as TENET tokens would be claimed on the snapshot only when supply is provided.
- It was informed that the contract was out of scope, and no more review was performed. Although found issues will be reported anyway.

5.6 AddressRegistry.sol

- Setter and getter based on the hash of a string.
- The setAddress could be checking if the address is not zero

5.7 TenetDepositor.sol

- The msg.value is not being checked for different from zero on createLockForUser, same for the increaseAmount function.

5.8 GaugeFactory.sol

- Missing zero checks on immutable values on constructor.
- The createGaugeProxyAndSubGauges can only be called by the governor address, extracted from the AddressRegistry. The cloneDeterministic will be using the hash of the lsd as the salt. There will be a loop of all _stakingTokens tokens and will create a subgauge deployment for it. All tokens must have the same LSD, it will skip the first token as it was already checked and taken as the reference for others.
- The proxy initialisers are correct and not missing any parameter.
- The createSubGauge will call addSubGauge, prior to verifying that the gauge proxy exists. This function checks if the token is already present. We can assume that lsdToGaugeProxy will make sure that the new staking token and the gauge proxy does use the same LSD.

5.9 GaugeProxy.sol

- Initialiser with address registry.
- The addSubGauge will verify if the token is already tracked and add it otherwise. Only the TLSDFactoryOrGaugeFactory can call the function.
- The removeSubGauge can only be called by the governance. In case that the function is called with an unknown _stakingToken loop, gas will be used unnecessary but no gauge or token will be removed.
- distributeToken will iterate over all stacking tokens, obtain the validator and update its activity based on the amount of signed blocks from the last iteration. However, if the same validator is in multiple stacking tokens data overriding does occur, this is already

reported as an issue. Rewards are correctly weighted and factored accordingly to the activity of the validator. The decimals of the staking token are used as the factor value to prevent divisions to resulting in 0 value. The decimal factor is removed when calculating the final rewardAmount value. The reward fees are transferred to the validatorFeeRecipient, previously validated and a max percentage of 10%. The remaining amount, min 90%, is transferred to the gauge and deposited as a reward using deposit_reward_token.

5.10 RewardVault.sol

- distributeRewards can only be called by the reward distributor. It does truncate the timestamp into day chunks. It will then compute the expected reward based on the dailyRewardRate. It will then transfer the reward tokens (tenet) to the caller, in this case the distributed reward contract. If the balance of the vault is less than the computed reward, only the total balance is sent, draining the vault.
  - In the case that multiple days are skipped without reward, the difference of those days will be used, and the reward rate used on the difference. It is not possible to set a new reward rate and cause missing rewards, as the set function will verify that the last pull of reward was performed on the current reward day.
- setDailyRewardRate can only be called if the last pull from the reward distributor was performed on the current reward day.

5.11 RewardDistributor.sol

- distribute and distributeMulti can be called by any sender.
- The internal _distribute function does obtain the gaugeType from the controller and gaugeAddr given as parameter.
  - If the timestamp is bigger than the last pull + 1 day, we call the distributeRewards on the vault. This will transfer the

TENET tokens to the RewardDistributor and stored under pulls for the current rounded timestamps (chunks of days).

5.12 VestFactory.sol

- The createVesting will clone a TenetVesting contract and a VestRewardReceiver and assign the vesting contract to its initialiser and connect the receiver with the vesting contract though the initialiser too. deployedVests is used as the nonce for the deterministic wallet creation.
  - The _startTime looks like it can be in the past. See if this is causing issues.

5.13 TenetVesting.sol

The vesting can be killed if created with the revocable parameter. The rescueFunds will only be callable if the kill was performed, only allowed by the governance. The

- The withdraw function will transfer to the beneficiary the calculated amount using withdrawable if the vesting is not killed/revoked.
  - The withdrawable function does calculate the amount vested since the last withdrawn. It will make sure that the total amount is used if the vesting period is already completed, and also make sure that the returned amount is never bigger than the balance of the vesting contract.
- The createLock, increaseLockDuration and increaseLockAmount does allow looking on the VotingEscrow contract the received tenet tokens. Increase the duration of the look, or increment the amount with a new approval.
- The delegateBoost allows delegating the ve boost to another address.
- The voteForGauge will set the gauge weight and only the beneficiary can call it.

5.14 VestRewardReceiver.sol

- The sendTokens function will verify if the vesting contract is killed. If so, the tokens will be transferred to the governance, otherwise to the beneficiary of the vesting contract.

THANK YOU FOR CHOOSING

HALBORN