Contracts vulnerabilities

Vulnerabilities list

Involved contracts and level of the bugs

The present document aims to point out some vulnerabilities in the contracts veOLAS, buOLAS, and VoteWeighting. Some of these vulnerabilities may lead to critical[1] bugs.

Vulnerabilities

1. `getPastVotes` function

Acknowledgments: This vulnerability was discovered thanks to howd4ys who kindly reported it by participating in the Autonolas Immunefi Bug Program.

Severity: Low[2]

In the veOLAS contract, the following function is implemented:

```
function getPastVotes(address account, uint256 blockNumber) public
view override returns (uint256 balance)
```

[1] The level of the bug is assigned by following the Immunefi classification

[2] Since no manipulation of governance voting can currently happen, this vulnerability identifies a smart contract that fails to deliver promised returns but doesn't lose value.

This function returns the voting power of the address `account` at a specific `blockNumber`.

This function has an incorrect behavior when the input `blockNumber` is smaller than the block number `n` where a lock was first created for the `account`.

Specifically, denoting by `T` the timestamp of the input `blockNumber` and `T1` the timestamp of the block `n`, the incorrect behavior arises because of the subtraction veOLAS.sol#L680 that becomes an addition when `blockTime=T` is smaller than `uPoint.ts=T1`. Denoting by `T2` the `endTime` for the locking created for the `account` at time `T1`, the calculation in veOLAS.sol#L680 provides the following value for the bias `bias=slope*(T2-T)`. However, the latter bias is bigger than the correct value for the bias at the timestamp `T1` which should be `slope*(T2-T1)`.

We recommend using as an input parameter of the function a `blockNumber` bigger than the block number `n` where a lock was first created for the `account`.

Note that the function `getPastVotes(account,blockNumber)` is used to weigh the voting power of users that cast a vote on a governance proposal. Whether there is a governance proposal and a user creates a lock after the beginning and no later than the end of the voting period, due to this vulnerability of `getPastVotes(account,blockNumber)`, the user would be able to cast a voting power bigger than its correct one. Note that the manipulation of the voting power has an upper bound due to the fact that there is a limited number of blocks in a voting period and that any locks last at least one week. Nevertheless, currently, there is no possibility of creating new locks, so this issue cannot affect any governance voting.

The wrapped veOLAS (wveOLAS) contract wraps original veOLAS view functions and serves as mitigating measures to address this issue.

2. `balanceOfAt` function

Severity: Low[3]

In the veOLAS contract, the following function is implemented:

```
function balanceOfAt(address account, uint256 blockNumber) external
view returns (uint256 balance)
```

[3] This function is not currently used in any of the Autonolas on-chain contracts, thus this vulnerability identifies a smart contract that fails to deliver promised returns but doesn't lose value.

This function returns the actual balance of the address `account` at a specific `blockNumber`.

This function has an incorrect behavior when the input `blockNumber` is smaller than the block number `n` where a lock was first created for the `account`.

As for the previous vulnerability explanation (`getPastVotes()`), the function `balanceOfAt()` uses the same binary search algorithm followed by identical value extraction, and thus returns a very first balance of a locked point, whereas it should return zero.

We recommend using `blockNumber` input parameter value bigger than the block number `n` where a lock was first created for the `account`.

The wrapped veOLAS (wveOLAS) contract wraps original veOLAS view functions and serves as mitigating measures to address this issue.

3. `_checkpoint` function

Severity: Medium[4]

In the veOLAS contract, the following function is implemented:

```
function _checkpoint(address account, LockedBalance memory oldLocked,
LockedBalance memory newLocked, uint128 curSupply) internal
```

According to the article on medium.com, the declaration of a memory struct *lastPoint* and its assignment to another memory struct leads to the pointer of the initial struct, and not its deep copy, that can be observed in line [219](219). This leads to the incorrect calculations of history points for the periods of time when there was no *checkpoint()* function called for more than a week.

However, there is more to the specified issue that leads to following observations:

- If the contract has not created a user point during a specific week, then when finally created, the internal checkpoint function writes a point in that week with the block number equal to the last created point but with a timestamp equal to the end time of the week that has just passed. If no points were created for several weeks,

[4] When voting via veOLAS, the incorrect value is returned as a read-only value, thus this could be declared as a Low severity. However, if there are consequences due to incorrect voting failure, then it is a potential damage to the DAO members, and then the severity is Medium.

then the internal checkpoint function recreates a point for each week of inactivity having the block number equal to the last created user point and the timestamp equal to the end time of the end of each skipped week.

- Even if the checkpoint is called once a week, two points are created: one point with a block number equal to the last created point but with a timestamp equal to the end time of the week, and another one with an actual block number and corresponding timestamp of the checkpoint call.

This behavior leads to the creation of supply points that have an incorrect block number detached from the actual timestamp. This further leads to the scenario where all supply points during the weeks of inactivity of veOLAS have the same block numbers as the first point that triggered the *checkpoint()*. In other words, all the supply points that were recreated at the end time of every week will not be correctly recovered during the block number search (via the block number itself or the timestamp). Any historic lookups between two supply points (not including points themselves) that were created immediately before and immediately later the exact end of a week or that were created with more than a week of inactivity will have an incorrect block number equal to the one of a first point.

This might potentially affect the voting functionality. If the voting was performed during the time that had to account for the inactivity weeks, immediately after the very last point before the end time of a week, or immediately after an eventful point was created at a the end time of a week, the weighted total supply (the overall number of votes) in the function *getPastTotalSupply()* **might** return incorrect values (depending on the first point with the same block number found via a binary search).

In the absence of deploying new contracts, we recommend running the analogue of the cron scheduler / service that checks for the veOLAS activity during the week, and if there was none, trigger a *checkpoint()* function call immediately before and immediately after end time of each week. This way, all the supply points will be updated throughout the time of the contract and, we increase the likelihood of having voting periods starting immediately after and before effective points with different blocks timestamps (not in the weekly time divider). As the protocol becomes more active, this issue will be minimized by the participation of DAO members.

To minimize a possible impact, the service triggering the *checkpoint()* call must be executed as close to the whole week of unix time as possible. Specifically, if the checkpoint is called at least once a week, a possible deviation in the total voting supply can only happen if the vote starts after the very last point of week (not in the weekly time divider) or before the very first point of a week. The supply deviation factor depends on the time difference between the very last weekly point (not in the weekly end divider) and the very first point after the weekly end time divider point.

Therefore, calling checkpoints as closer to the end and the beginning of the week of unix time as possible the supply deviation can be minimized. A probabilistic analysis of how likely such a scenario can happen is out of the scope of this document. However, despite its likelihood, it is worth mentioning that, even in such a scenario, there is no certainty that the wrong point will be picked by the binary search and ultimately there is no certainty that the issue is going to affect the expected result.

4. `createLockFor` function

Severity: Medium

In veOLAS and buOLAS contracts, the following function is implemented:

```
function createLockFor(address account, uint256 amount, uint256
unlockTime) external
```

This function allows anyone, even a smart contract, to create a lock for a third-party account. If the third-party account has already a locked amount the call will be reverted. If not and the OLAS amount provided as input is non-zero then a lock is created.
As a consequence, any third-party account can be forced into a long lock length (for a maximum of 4 years for veOLAS and 10 years for buOLAS) by an attacker calling `createLockFor` with a very small amount of OLAS (i.e. 1/10 ** 18) and a max lock length. An attacker could use this to prevent locks over a given adversarially chosen interval by front-running all locks in this manner. All accounts with an intent to lock for less than 4 years would be affected. We assign a low likelihood to this attack, as it is not economically profitable for the attacker.

Indeed, the caller of the `createLockFor` function can lock for third-party users only by using its own OLAS tokens. So the mintable OLAS tokens can be temporarily frozen only with an attacker's extensive cost.
In the buOLAS contract, there is also an extra guardrail that can be considered. If the attack has been discovered, it is possible to invoke a governance vote to revoke the unvested OLAS of the third-party account that has been forced in a long lock into buOLAS. If the governance approves the revoke, the third-party account can call the buOLAS withdraw function, and all non-vested OLAS tokens will be burned. When the withdrawal function is called less than one year after the attack, all the contract status can return to their original status before the attack has been made.

5. `totalSupplyLockedAtT` function

Severity: Low

In the veOLAS contract, the following function is implemented:

```
function totalSupplyLockedAtT(uint256 ts) public view returns
(uint256)
```

The function is used solely by the `totalSupplyLocked()` function with the current `block.timestamp`. By the original design, it is not intended to have a `ts` parameter smaller than the current `block.timestamp`.

We recommend not to call this function for any external purposes. It is a view function that is not currently used externally in any of Autonolas on-chain protocol contracts, and thus does not affect any intended behavior.

The wrapped veOLAS (wveOLAS) contract wraps original veOLAS view functions and serves as mitigating measures to address this issue.

6. `getPastTotalSupply` function

Severity: Low

In the veOLAS contract, the following function is implemented:

```
function getPastTotalSupply(uint256 blockNumber) external view
returns (uint256)
```

The function returns the voting power of a specified block number. However, by the original implementation, the requested block number must be at least equal to the zero supply point block number, or the block number of a contract deployment. Otherwise, the function reverts instead of returning a zero value.

We recommend not to call this function with the input block number value less than a zero supply point block number, since it is meaningless anyway as there must be no values before the very first supply point is created in the contract.

7. `processMessageFromForeign` function

Severity: Informative

In the HomeMediator contract, the following function is implemented:

```
function processMessageFromForeign(bytes memory data) external
```

The role of HomeMediator contract is to execute actions based on governance proposals originating from Ethereum. This execution is rigorously bound to governance decisions, with the validation of the message sender being restricted to the Timeloch address on Ethereum.

In the current implementation, the `processMessageFromForeign()` method ensures that the `msg.sender` aligns with the Ethereum Timelock address. However, it does not enforce a verification of the source `chainId` to match Ethereum's `chainId`. This poses no immediate issues as the arbitrary message bridge contract, facilitating communication between Ethereum and Gnosis, exclusively processes requests from the Ethereum chain.

For future scenarios where the arbitrary message bridge contract might handle requests from diverse chains (as outlined in the doc), it is recommended to improve the implementation of HomeMediator's `processMessageFromForeign()` method by incorporating a `chainId` check.

8. `removeNominee` function

Severity: Informative

In the VoteWeighting contract, the following function is implemented:

```
  function removeNominee(bytes32 account, uint256 chainId) external
```

The `removeNominee()` function is designed to remove a nominee from the system. This operation is restricted to the contract owner.

Whether the nominee exists, the nominee's current weight is then set to zero for the next checkpoint time. The total weight sum is updated to reflect the removal of the nominee's weight. If a dispenser contract is configured, the function calls the removeNominee method on the dispenser to ensure this is aware of the removal.

If a nominee is removed and users have allocated non-zero weight to that nominee, the associated voting power becomes orphaned. The contract doesn't automatically retrieve or reallocate user voting power within the `removeNominee()` function itself. However, users can reclaim their voting power using the `retrieveRemovedNomineeVotingPower()` function. It's advisable for voters to update a non-zero weight of their nominee to zero before the nominee's removal is expected to happen or to reclaim their vote after the nominee's removal has occurred.

Additionally, when a nominee is removed, the last nominee in the set will take the place of the removed nominee, changing its ID at the end of the **removeNominee()** call. It's important to note that the same ID can correspond to different nominees at different times, depending on removals.

9. `_addNominee` and `removeNominee` functions

Severity: Informative

In the VoteWeighting contract, the following functions are implemented:

- `function _addNominee(Nominee memory nominee) internal`
- `function removeNominee(bytes32 account, uint256 chainId) external`

The `removeNominee()` function is designed to remove a nominee from the system, and this operation is restricted to the contract owner. On the other hand, `_addNominee()` adds a nominee to the system.

In both cases, if a Dispenser is configured, these functions respectively call the `addNominee()` and `removeNominee()`methods on the dispenser contract. This ensures that the dispenser is kept informed about any nominees being added or removed.

It's highly recommended for the deployer to set up a dispenser contract immediately after deploying VoteWeighting and ensure that no nominees were added or removed before that. This precaution helps prevent potential synchronization issues between nominees on the VotingWeight and Dispenser contracts.

10. `voteForNomineeWeights` function

Severity: Informative

In the VoteWeighting contract, the following functions are implemented:

```
function voteForNomineeWeights(bytes32 account, uint256 chainId,
uint256 weight) public
```

The function is designed to allocate voting power for changing pool weights. Note that following the original Curve implementation, the function reverts if the next time of recording weights is equal to the end of veOLAS lock (reverts if `nextTime >= lockEnd`). In case of equality, the voting account is not able to place their weights as their veOLAS lock expires in a week. We consider this not to be an issue, as one week of time results in a very low voting power addition, and the lock can always be extended for longer in order to make a bigger weighting input.