Contracts vulnerabilities

Vulnerabilities list: #1

Involved contracts and level of the bugs

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

Vulnerabilities

1. `_getPastVotes` vulnerability

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

Severity: Low level[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.

We will soon address this issue in order to always return a correct calculation of the voting power of the `account` at any `blockNumber`.

2. `_checkpoint` vulnerability

Severity: Medium[3]

In the veOLAS contract, the following function is implemented:

[3] 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.

```
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. 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.

This behavior leads to the creation of supply points that have an incorrect block number which is 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 for the weekly periods of veOLAS inactivity 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 in the *_checkpoint()* function 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, 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. This way, all the supply points will be correctly updated throughout the time of the contract. As the protocol becomes more active, this issue will resolve itself by the participation of DAO members.

3. `createLockFor` vulnerability

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 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' s withdraw function, and all non-vested OLAS tokens will be burned. When the withdraw function is called less the one year later the attack, all the contract status can return to their original status before the attack has been made.

4. `createBalanceFor` vulnerability

Severity: Critical

The owner of the Sale contracts can lock the OLAS in the sale account by creating a veOLAS and/or a buOLAS balance for any address. Once a balance is created for an account, the account owner's private key(s) is (are) the only one(s) that can claim its balance. So if a balance is erroneously created, the OLAS in the sale account would be rendered unusable.

Furthermore, any given address can only claim exactly once on the Sale contract. Consider the scenario, where an account has a balance created via `createBalanceFor`, then the account claims, then another balance is created. The second time the account cannot claim due to the fact that both in veOLAS and buOLAS in order to `createLockFor`for an account its locked balance must be zero leading to stuck funds.

Finally, an attacker can acquire OLAS and create a very small lock with `createLockFor` into the veOLAS or buOLAS contracts for any account with a balance to claim in Sale. This would make all these balances stuck.

We recommend no further usage of the Sale contract and its deprecation after the Autonolas private sale.

Comments

The following are not vulnerabilities. These are properties of the code that are not necessarily wanted in the design but that do not lead, as far as we can tell, to any attacks.

1. `depositFor` comment

In veOLAS contract, the following function is implemented:

function depositFor(address account, uint256 amount) external

This function allows anyone, even a smart contract, to increase the locking amount for a third-party account. If the third-party account has not 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. So the third-party account can be forced by an attacker into a larger lock amount (potentially all mintable OLAS that can be acquired and locked for the third-party account).

We don't think this qualifies as an attack/vulnerability. It would just be an issue whether no one has OLAS to sell (a very unlikely event), but there's still the utility of the system