BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT:

defi.money
boosted-staker

June 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Defi.Money - BoostedStaker
Website	defi.money
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/defidotmoney/boosted-staker/tree/f9389f1dd794b779ea56fea41ec599cdf8ac5503/contracts
Resolution 1	https://github.com/defidotmoney/boosted-staker/tree/e860d061a5baa411e87b735a993252833b858273

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	3	3		
Medium	4	1		3
Low	2		1	1
Informational	1			1
Governance	0			
Total	10	4	1	5

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

BoostedStaker

The BoostedStaker contract is a custom staking contract which allows users to stake and/or lock their tokens. The goal of this contract is to produce a specific weight for each user depending on their stakes/locks. The weight is produced by the base token amount which is staked/locked and an additional multiplier which is determined as MAX_WEIGHT_MULTIPLIER and can then be used in third-party contracts for different purposes, such as voting power, for example.

For instance, if users lock their tokens, these can only be withdrawn once the locking period has passed (STAKE_GROWTH_EPOCHS) but they will immediately receive the base amount + (baseAmount * (MAX_WEIGHT_MULTIPLIER-1)) as weight.

Contrary to that, users that just stake their tokens without a lock, will be able to withdraw their stake anytime but will only receive the base amount as weight and then each progressed epoch a pro-rata share of (baseAmount * MAX_WEIGHT_MULTIPLIER) depending on the size of STAKE_GROWTH_EPOCHS until the stake has fully matured.

Whenever an epoch progresses, there are two distinct checkpoints:

a) Checkpoint per individual user
b) Checkpoint for the global supply

The user checkpoint is updated whenever a user stakes or unstakes and it will simply loop over all epochs until the current system epoch, since the last user update. During these loops the user's weight is incremented based on the total pendingStake with each epoch. Additionally, once an epoch's stake is considered as matured (once the epochToRealize is reached), the full pendingStake and lockedStake for this specific epoch is translated to the realizedStake, marking it as fully weighted and ready to withdrawal, which will then lower the future weight increase because this particular pending amount will not further contribute to the weight increase.

The global checkpoint works with the exact similar mechanism, by increasing the global weight with a pro-rata share of the aggregated pendings of all users, while decreasing the aggregated pending once a user's pending is due. This will ensure that the aggregated weights of all users and the global weight is always equal.

Below we will explain each mechanism in-depth:

Stake mechanism: There are two possible scenarios: Lock and Stake

a) Lock mechanism: Users can lock their tokens which will grant them immediately full weights. Once tokens are locked these can be withdrawn only STAKE_GROWTH_EPOCHS later, which is handled during the Stake expiration mechanism.
b) Stake mechanism: Users can stake their tokens and will immediately receive a base weight with the additional privilege that the weight is linearly increased during each epoch until STAKE_GROWTH_EPOCHS have been surpassed.

Unstake mechanism:

Whenever an unstake happens, the user's specific checkpoint is updated which means that the most recent matured stake (pending/locked) is eventually translated into the realizedStake and the weight is increased. After this state has been updated, a deterministic approach fetches the least progressed stake from a user and then deducts the desired withdrawal amount from this stake.

This is done by looping over all stakes and fetching the most recently created stake, which could even be a stake which has just been created during the recent epoch. If that stake size is sufficient to cover the withdrawal needs, the withdrawal will further progress, if not, the next existing stake is fetched and so on. If all pending stakes have been fetched but the withdrawal amount is still not met, the realizedStake is then used, which consists of fully progressed stakes and expired lock stakes.

The reason behind this approach is to not accidentally penalize users by withdrawing more progressed stakes, as this would essentially result in a larger weight decrease.
The weight is decreased by the exact weight which has been accumulated by the withdrawn amount. If for example only 50% of the base amount has been withdrawn and there have been 3 epochs of weight increases, then only 50% of the base weight and 50% of the 3 epoch increase value is deducted from a user's weight.

Individual Weight Increase:

Whenever an epoch progresses, the checkpointAccount function can be invoked which loops from the lastUpdateEpoch to the current systemEpoch. During each loop the following things happen:

a) Weight is increased: The pendingStake amount represents the aggregated amount of all pending stakes from a user. These pending stakes will linearly increase a user's weight whenever an epoch has been surpassed.

b) Epoch stake is discarded: As above mentioned the pendingStake amount is the aggregate of all pending stakes for each epoch. If an epoch stake has reached its realizedEpoch, this means the epoch's pending stake has been fully allocated to the user's weight and should not further increase it. Therefore this discarded epochs stake is removed and allocated to the realizedStake, indicating it has been fully allocated or unlocked.

Global Weight Increase:

In a similar fashion to the individual weight increase is the global weight increase handled. The target of this logic is to always reflect the aggregated weight of all users in the globalEpochWeights mapping.
This is done by aggregating all pending stakes in the globalGrowthRate variable which will then just increase the global weight in a similar fashion as described above. Additionally whenever the realizedEpoch from an individual is reached, this will also be reflected in the global state using the globalEpochToRealize variable, decreasing the globalGrowthRate by the to be removed pending amount.

Stake expiration mechanism:

As already explained above, whether a normal stake which increases a user's weight over time or a locked stake, once the specific threshold of STAKE_GROWTH_EPOCHS has been surpassed, the pendingStake as well as lockedStake amount is erased and will be transferred 1:1 into the realizedStake, allowing a user to withdraw the full amount. Furthermore the stake is discarded from the system.

Appendix: Bitmap Usage:

The contract uses a storage-friendly mechanism for stake/epoch correlation. Whenever a user invokes the stake function, a 1 is pushed to the rightmost bit on the bitmap variable (updateEpochBitmap), which keeps track of how many epochs have been updated since the initial deposit. For each epoch update, the bitmap is shifted one to the right until the deposit has reached its maturity epoch. This is determined by STAKE_GROWTH_EPOCHS and the corresponding MAX_EPOCH_BIT variable. With this approach it is possible to handle multiple deposits in different epochs without any manipulation.

Illustrated:

A user has invoked the stake function during epoch 1, this will set the bitmap to 0000000000000001 (binary representation). STAKE_GROWTH_EPOCHS is 15 and MAX_EPOCH_BIT is 1<<15.

For each epoch update the bitmap is shifted 1 to the left which will result in the bitmap 1000000000000000 after 15 epochs (if no other deposits were made). In that scenario, the deposit is deemed as matured and the necessary interaction is executed. Whenever an epoch is updated, there is a bitwise AND operation between the bitmap and the MAX_EPOCH_BIT (mask). If this operation results in MAX_EPOCH_BIT, it indicates that the stake has reached its maturity epoch.
In another scenario, if for example a deposit is made in the 14th epoch, the bitmap will look as follows: 1000000000000010. The corresponding bitmap index is removed once a stake is withdrawn.

Privileged Functions

- transferOwnership
- renounceOwnership
- disableLocks
- sweep

Issue_01	Lack of _epoch validation within checkpointAccountWithLimit function allows user to manipulate account epoch
Severity	**High**
Description	Currently, it is possible to update an account state via the checkpointAccountWithLimit function, passing a custom _epoch parameter. The idea of this function is to update an account state only up to a certain epoch. However, there is currently no validation that the provided _epoch is in fact in the future. A malicious attacker can therefore provide an _epoch parameter which is older than the latest updated epoch, disrupting the account data algorithm.
Recommendations	Consider simply ensuring that the provided _epoch is not older than lastUpdateEpoch.
Comments / Resolution	Resolved.

Issue_02	Incorrect bitmap flipping during unstake will result in permanently locked funds
Severity	**High**
Description	First of all, we need to understand that unstaking tokens will influence two form of stakes: a) pendingStake: These are normal, unlocked stakes that are withdrawable anytime. They currently do not grant users their full weight because they are still in the progress mode. b) realizedStake: These are the already expired locked and normal stakes, they grant users their full weight because they are fully progressed To not negatively impact a user's weight during a withdrawal, the unstake function processes withdrawals by prioritizing the most recent stakes first, followed by older stakes. This minimizes the negative impact on the user's weight. During each loop, the pendingStake for the specific epoch is decreased, if this epoch's stake does not fully cover the desired amount, another loop is executed which then decreases the pendingStake from the next epoch and so on and so forth. In the special case where the leftover amountNeeded is exactly covered by the pendingStake of an epoch, the bitmap is manipulated which will effectively discard the epoch's stake (ie. making it unusable): https://github.com/defidotmoney/boosted-staker/blob/f9389f1dd794b779ea56fea41ec599cdf8ac5503/contracts/BoostedStaker.sol#L392 This operation is fatally wrong as it also discards epoch stakes with an existing lock, effectively making these tokens unretrievable.
Recommendations	Consider checking if the epoch's stake has a lock and only discard it if there is no existent lock.
Comments / Resolution	Resolved.

Issue_03	Codebase is vulnerable to silent overflows on multiple occasions
Severity	**High**

Description	The codebase incorporates a gas-optimizing approach which uses smaller uint types than uint256. Notably as a safeguard, it is not possible to deposit an amount with type uint112.max:
	
	https://github.com/defidotmoney/boosted-staker/blob/f9389f1dd794b779ea56fea41ec599cdf8ac5503/contracts/BoostedStaker.sol#L512
	
	This safeguard seems to protect against most silent overflows. However, there are still multiple occasions in this codebase where a silent overflow can happen, due to unsafe casting. (Or a revert due to solidity's built-in overflow protection, incase of arithmetic operations)
	
	Most notably the following example:
	
	uint112 globalWeight = uint112(_checkpointGlobal(systemEpoch))
	
	In any scenario, a single deposit must not be as high as uint112. However, multiple deposits can exceed that threshold. This is even assumed in the following line:
	
	totalSupply += uint120(_amount);
	
	Since totalSupply is from type uint120, the contract is meant to be compatible with tokens of such a large supply.
	
	However, as already highlighted above, the globalWeight does not support this. Since the globalWeight is basically the supply "inflated", it can be > uint112. In such a scenario it will silently overflow, downcasting the globalWeight to an incorrect value.
	
	One can even see here:
	
	globalEpochWeights[systemEpoch] = uint128(globalWeight + weight);
	
	The weight is assumed to be from type uint128, not uint112.

	This will essentially break the whole global weight calculation, rendering the contract useless. There are also other spots in the codebase that are vulnerable to this issue.
Recommendations	Consider either ensuring that no tokens with a large supply are used or increasing the uint types throughout the codebase
Comments / Resolution	Resolved, a check was implemented which ensures that the total staked amount cannot be >= uint112. Furthermore the unsafe casting to uint112 has been adjusted to uint128.

Issue_04	Lack of grace period between stake and unstake allows for flash-setting weight
Severity	**Medium**
Description	Since this contract is likely to represent voting power or something similar, the stake design is inherently incompatible for such a purpose. Due to the fact that users will immediately receive their base amount as weight and can withdraw in the same epoch, malicious users can simply stake, abuse their present weight and then withdraw immediately afterwards.
Recommendations	Consider implementing a grace period which does not allow withdrawal in the same/next epoch. This approach could be achieved by incorporating the following changes to the unstake function: a) Adjust the require statement to exclude the pendingStakes from the accountEpochToRealize mapping for the current and the previous epochs b) Start with the initial mask shifted two to the left (start the loop with epochIndex = 2) This approach should sufficiently exclude the current and pre-current epochs from the withdrawal process.

	HOWEVER, due to the complexity of the algorithm and the intrusiveness, such a change can impact the broader context and therefore it must be carefully validated. Such a validation will take more time than a usual resolution round, thus may result in a small, additional fee.
Comments / Resolution	Acknowledged, this is no issue for the use-case of this contract.

Issue_05	disableLocks call can be frontran
Severity	**Medium**
Description	This contract exposes a disableLocks function which allows the owner to set locksEnabled to false. Once this is set to false, it is not possible to lock tokens anymore and all locks are considered as unlocked, translating them into a withdrawal (realized) stake, while keeping the weight at its maximum. This execution can be frontran by malicious users, which results in these users gaining maximum weight while being able to withdraw their stake without any limitations. The same issue exists in the Factory contract and should be fixed there as well.
Recommendations	Consider implementing a grace period, such as calling the “announceUnlock” function which then allows calling the “disableLocksGlobally” function only 1 day later. Using this approach will give all users a fair chance to lock their tokens beforehand and the maximum weight without the actual locking disadvantage.
Comments / Resolution	Acknowledged, this function will only be used for sunsetting purposes. Frontrunning will not grant any benefits.

Issue_06	checkpointGlobal function can run out of gas
Severity	**Medium**
Description	Currently, the only function available to update the global epoch state is checkpointGlobal. In contrast, the account state can be updated via both the checkpointAccount and checkpointAccountWithLimit functions. In the worst case scenario, when iterating over an excessively large number of epochs, checkpointGlobal can run out of gas, permanently DoS'ing the contract.
Recommendations	Consider implementing a checkpointGlobalWithLimit function.
Comments / Resolution	Resolved.

Issue_07	Unstake function is vulnerable to underflow revert
Severity	**Low**
Description	Whenever users withdraw their tokens, several arithmetic operations are executed. Most are simple subtractions. Whenever a subtraction happens, one should become vigilant as this exposes a potential underflow risk. In staking contracts which allow withdrawing an arbitrary amount, such a risk is negligible because one can simply withdraw: (amount - 1 wei) to circumvent such an issue. This is a totally different topic in staking contracts which only allow withdrawing the full amount at once. This contract is a mix of both. The unstake function is written in such a manner that it attempts to withdraw from the current epoch first and then loops over all following epochs until the desired withdrawal amount is met. This is exactly where the problem lies, as the loop potentially attempts to withdraw the full amount: https://github.com/defidotmoney/boosted-staker/blob/f9389f1dd794b779ea56fea41ec599cdf8ac5503/contracts

/BoostedStaker.sol#L383

This paves the way for such an issue to occur because users cannot simply “leave 1 wei” in that position. The position must be completely cleared to allow for withdrawing the next iteration.

Now let’s inspect all subtractions during unstaking:

```
globalEpochToRealize[epochToCheck] -= pending;
amountNeeded -= pending;
```

OR

```
epochToRealize[epochToCheck].pending -= amountNeeded;

globalEpochToRealize[epochToCheck] -= amountNeeded;

acctData.realizedStake -= uint112(amountNeeded);
```

OR

```
acctData.pendingStake -= uint112(pendingRemoved);

globalGrowthRate -= uint112(pendingRemoved);
globalEpochWeights[systemEpoch] -= uint128(weightToRemove);
uint256 newAccountWeight =
accountEpochWeights[_account][systemEpoch] - weightToRemove;
```

All green highlighted operations are safe because they use the nominal amount. The red operations however are potentially unsafe because the subtraction amount undergoes an arithmetic operation before which calculates the weight.

For these operations it is possible that they round down/don’t round down depending on the amounts which are part of the operation (it can be a clear division or a division with precision loss)

In such a scenario, it is possible for the execution to revert because the subtraction amount can become larger than what was initially

allocated to the state variables.

*Interestingly, a comment indicates that this issue is known:

// division before multiplication is intentional to ensure consistent rounding loss each epoch// otherwise there is a possibility for an underflow when the last user unstakes

However, the small edge-case where there are divisions without precision loss vs divisions with precision loss can occur, was not accounted for.

We acknowledge that such an issue can happen during a loop seems rather from a theoretical nature. However, there is still the issue left which prevents users from withdrawing their whole stake:

Illustrated:

For example let's say that `MAX_WEIGHT_MULTIPLIER == 2`, `STAKE_GROWTH_EPOCHS == 10` and there are 2 lock actions with the following deposit amounts:

`amount1 = 103`
`amount2 = 107`

For both deposits, `amount / STAKE_GROWTH_EPOCHS` rounds down, so the weight added will be _less_ than a situation where there is no rounding error

After stake is called twice, the user tries to unstake their entire balance. Their total `amount` is `210`.

`amount = 103 + 107 = 210`

Note that `amount` (210) is divisible by `STATE_GROWTH_EPOCHS` (10). Therefore, no rounding loss occurs during unstake.

Since rounding loss for the amounts happened during the 2 function calls to `stake`, and no rounding happened during `unstake`, the weight calculated during `unstake` will be greater than the total weight

	calculated while staking. Therefore, the line `uint256 newAccountWeight = accountEpochWeights[_account][systemEpoch] - weightToRemove;` will underflow and thus revert.
Recommendations	Consider just setting these state variables to zero in a scenario where the amount to subtract is larger than the state variable itself.
Comments / Resolution	Partially resolved, this has been fixed for the individual weight but not for the global weight. Note - here is some analysis which concludes that the global weight will never overflow. However, we do recommend adding the check for global weight for safety. - Sum of new account weights always == global weights for an epoch - the same weight is **always** subtracted from the account weight of an epoch and global weight - therefore, since the last withdrawer has global weight == their account weight, and weightToRemoved is capped by the account weight, it is impossible for the subtraction of global weight to underflow.

Issue_08	Locking or staking at the end of an epoch will grant strategic advantage
Severity	**Low**
Description	Currently, it is possible to stake/lock one block before an epoch has passed. This will grant users a strategic advantage because in the next block the progress of the stake is already increased.
Recommendations	There is no simple fix for this issue as it is an inherent part of the contract design.
Comments / Resolution	Acknowledged.

Issue_09	Weight deduction rounds down during withdrawals
Severity	**Informational**
Description	During smart contract development it is crucial to always round against the favor of the user. This is a known technique to prevent any exploit that can be the result of a potential rounding issue. WIthin the contract, the _getWeight function is used during stake and during unstake. It is important to mention that this function strictly rounds down, which is correct in the scenario of a stake. For an unstake, it is however incorrect as this means it will decrease the weight less (due to the calculation rounding down). We just raise this issue for acknowledgement purposes due to the following reasons: a) Changing it and rounding up during unstake would mean that an underflow is provoked which can results in DoS b) Due to the business logic of this contract and the weight being generally large compared to any potential gain, this issue is negligible.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Factory

The Factory contract is a simple factory which is responsible for the deployment of the BoostedStaker contract.

It is using the deterministic deployment approach via create2 and a corresponding salt which allows anyone to determine an address before it has been deployed. Furthermore, the owner can deploy different BoostedStaker contracts with different MAX_WEIGHT_MULTIPLIER's.

Privileged Functions

- transferOwnership
- renounceOwnership
- acceptOwnership
- deployBoostedStaker
- disableLocksGlobally

Issue_10	disableLocksGlobally can be frontrun
Severity	**Medium**
Description	In a similar fashion as already described within the "disableLocks can be frontrun" issue, this issue can be abused.
Recommendations	Consider following the same fix approach as explained in "disableLocks can be frontrun".
Comments / Resolution	Acknowledged, this function will only be used for sunsetting purposes. Frontrunning will not grant any benefits.