Sushi Swap

BentoBox Strategies and Staking Contract

Smart Contract Security Review

Version: 2.0

June, 2022

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Sushi smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Sushi smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Sushi smart contracts.

Overview

BentoBox can allocate a percentage of a token's reserves to be used in an underlying strategy to generate some passive income. **BentoBox Strategies** is a set of smart contracts showcasing the use of these investment stategies. The project defines a `BaseStrategy` interface as well as several actual strategy implementations. For example there are investment strategies using `SushiSwap` itself or the popular lending platform `Aave` across multiple networks.

StakingContract implements a versatile and permissionless ERC20 token staking system. Anyone can create a new staking "incentive" for a custom period of time by depositing arbitrary tokens as reward. Users can then lock up the respective token and earn a share of these rewards in return. Every stake can be subscribed to up to six different incentives at the same time.

Security Assessment Summary

This review was conducted on the files hosted on the Sushi repository and were assessed at commit af27913 for Bentobox Strategies and 4804391 for Staking contracts.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts: specifically their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 12 issues during this assessment. Categorized by their severity:

- Critical: 2 issues.
- High: 3 issues.
- Medium: 1 issue.
- Low: 1 issue.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Sushi smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
SSBS-01	Reentrancy Vulnerability Allows Draining All Funds	Critical	Resolved
SSBS-02	Missing Input Validation Allows Subscribing to Non-Existent Incentives	Critical	Resolved
SSBS-03	`startTime` Incentive Restrictions Can be Bypassed	High	Resolved
SSBS-04	Exiting Aave May Result In Invalid Accounting in BentoBox	High	Open
SSBS-05	Non-standard ERC20 Tokens are Not Supported	High	Open
SSBS-06	Unstaking `stkAave` Extends Cooldown Time	Medium	Open
SSBS-07	Missing Input Validation	Low	Open
SSBS-08	`batch()` Incorrectly Interprets Error Messages	Informational	Resolved
SSBS-09	Using `batch()` to Unsubscribe from Many Incentives Could Lead to Unexpected Results	Informational	Closed
SSBS-10	Opportunities for Malicious Tokens to Poison Logs	Informational	Closed
SSBS-11	Contracts Do Not Implement Safe Ownership Transfer Pattern	Informational	Open
SSBS-12	Miscellaneous `BentoBox Strategies and Staking Contract` General Comments	Informational	Open

SSBS-01	Reentrancy Vulnerability Allows Draining All Funds		
Asset	StakingContractMainnet.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A reentrancy vulnerabilitiy in the function `stakeToken()` allows an attacker to drain the funds of any ERC20 token deposited in the contract.

In `stakeToken()` on line [180], `msg.sender`'s liquidity is updated in the state variable `userStakes`, however the incentive's total liquidity is not updated until line [202]. In between, on line [194], there is a call to `_claimReward()` which passes execution flow back to the `token` being transferred. Using a malicious `token` that can react to transfers, such as an ERC777 token, or a custom attack token, the attacker can reenter the contract in between these two lines and interact with the contract in a partially updated state.

In the partially updated state, `userStake.liquidity` has been increased but the total liquidity of one or more incentives have not been. `userStake.liquidity` is global across all the user's incentives, and is used as a multiplier when rewards are calculated. Therefore, a malicious user may multiply the rewards for unclaimed incentives by an inflated figure, and drain tokens.

The steps taken for this attack are as follows, suppose that there are multiple incentives where USDC is the staking token. Bob is the attacker and has created a malicious token contract, ATT.

1. Bob creates an incentive staking USDC for rewards in ATT.
2. Bob deposits some USDC into multiple target incentives and also into his ATT incentive. All target incentives must be staking USDC for some other token. It is these other tokens that will be drained. The order of subscriptions is also important. The ATT incentive needs to be first.
3. Bob waits for some rewards to accumulate.
4. Bob takes a flash loan of USDC and calls `stakeToken()` to deposit the flash loan with the parameter `transferExistingRewards` as `True`. As the staking contract loops through the incentives that Bob is subscribed to on line [184], it calls the ATT incentive first (as Bob has been careful to subscribe in the correct order for this to happen).
5. When ATT's `safeTransfer()` function is called, it passes execution control to Bob, allowing reentrancy. Bob calls `claimRewards()` for the other incentives he is subscribed to. The reward multiplier `usersLiquidity` on line [378] will be out of proportion to the overall liquidity and this can result in the staking contract paying out its entire balance of the reward token.
6. Bob calls `unstakeToken()` to get back the flash loan and repays it.

A similar reentrancy vulnerability occurs using `_claimReward()` which instead reenters the function `unsubscribeFromIncentive()` and may overflow the `unchecked` operation on the following line.

```
unchecked { incentive.liquidityStaked -= userStake.liquidity; }
```

Recommendations

There are two preventive measures that may be taken to mitigate reentrancy:

1. Carefully implement the *checks->effects->interactions* pattern throughout `StakingContractMainnet`. In particular, make sure that it is not being violated within function calls. This ensures that all external calls are made after state updates.
2. Use a reentrancy guard, the best known of which is OpenZeppelin's `ReentrancyGuard`, and apply its modifier `nonReentrant()` on all `public` and `external` functions.

This reentrancy protection will also prevent against the `unchecked` overflow on line [**286**]. In that case, consider also removing the `unchecked` wrapper to allow for overflow protection in the two functions `unsubscribeFromIncentive()` and `unstakeToken()`. The gas increase of using checked math is small for these two variables and the added security significant.

Resolution

This issue was resolved by using the `Solmate` re-entrancy guard over the required `public` and `external` functions.

The fix is outlined in PR #2.

SSBS-02	Missing Input Validation Allows Subscribing to Non-Existent Incentives	
Asset	`StakingContractMainnet.sol`	
Status	**Resolved:** See Resolution	
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Missing input validation checks allows an attacker to steal substantially more rewards than available in a given incentive.

The attack occurs due to a bug which allows subscribing to incentives which do not yet exist. Furthermore it is possible to subscribe to an incentive multiple times if it does not yet exist. `subscribeToIncentive` does not check if the user-supplied `incentiveId` is actually valid.

The bug is exploited by subscribing to a nonexistent incentive multiple times as seen in the following scenario:

1. The attacker subscribes 6 times to a nonexistent, but upcoming, incentive number. This is done using a fake zero-address subscription.
2. A genuine user creates a new incentive with X rewards in USDC. The incentive number matches the previously nonexistent one our attacker subscribed to.
3. As other users create similar USDC based rewards, the contract begins to hold a lot of USDC.
4. The attacker fakes their zero-address subscriptions for 6 real ones by unsubscribing and re-subscribing. He proceeds to stake some amount.
5. After some time has passed, The attacker unstakes some strategic amount, choosing save rewards. This will set their `rewardPerLiquidityLast` erroneously low.
6. The attacker calls `claimRewards()` for the victim's `incentiveId`. They have the ability to drain more rewards than those provided by the incentive creator, draining funds from other users.

Note that it is also possible to subscribe to the incentive with ID zero, which is never used.

Recommendations

This issue may be mitigated by preventing the subscription to incentives which have not yet been created. This can be done by adding a check that forbids subscribing to an incentive which has a `creator` value of the zero-address.

Alternatively this issue may be mitigated by ensuring `0 < incentiveId && incentiveId <= incentiveCount`.

Resolution

This issue was rectified by rejecting subscriptions to incentives which have not been created. The additional check can be seen in the following code.

```
if (incentiveId > incentiveCount || incentiveId <= 0) revert InvalidInput();
```

Details of the patch can be seen in PR #2.

SSBS-03	`startTime` Incentive Restrictions Can be Bypassed		
Asset	`StakingContractMainnet.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The conditional statement in `_accrueRewards` updates the `lastRewardTime` for incentives. However, it will overwrite the `startTime` for incentives which haven't started yet.

As the `liquidityStaked` for new incentives is initialised with zero, their `lastRewardTime` gets updated as well. As a result, it overwrites the value of `lastRewardTime` to `block.timestamp`.

The impact is that future calls to `_accrueRewards()` will perform rewards calculations based on an incorrect `lastRewardTime`. Users can accrue and withdraw incentives after `_accrueRewards` has been called the first time, bypassing the initial `startTime`.

Recommendations

An additional statement may be added to `_accrueRewards()` which will only update `incentive.lastRewardTime` if `block.timestamp > incentive.lastRewardTime`.

Resolution

The development team has implemented the following check in `_accrueRewards()` to ensure the `lastRewardTime` is only updated after the start time.

```
} else if (incentive.liquidityStaked == 0 && lastRewardTime < block.timestamp) {
    incentive.lastRewardTime = uint32(maxTime);
}
```

The fix is outlined in PR #2.

SSBS-04	Exiting Aave May Result In Invalid Accounting in BentoBox		
Asset	AaveStrategy.sol		
Status	Open		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The Aave protocol lends out deposited funds and so it is possible to have a situation where the balance of `strategyToken` in Aave is less than that owed to the BentoBox strategy. If there is insufficient balance the BentoBox strategy will only withdraw the available balance.

The deficit between the deposited amount and withdrawn amount will be considered a loss in BentoBox when calling `setStrategy()`. The loss will cause the value of BentoBox shares to decrease.

The amount remaining in the Aave protocol may be withdrawn by the admins at a later time when there is sufficient tokens in the protocol to accept a withdrawal. However, the shares would need to be redistributed to users manually by taking a snapshot of the user share balances at the time the exit was made. This is highly impractical as there is a large number of users requiring a large distribution of tokens. Furthermore, during this process, admins will have full control over the funds, creating a centralisation risk.

A similar issue occurs during `_exit()` since the `withdraw()` external call is wrapped in a `try-catch` statement. If the external call fails the entire balance will remain in the Aave pool. The deficit in this case will be the entire amount deposited and the loss will be distributed upon BentoBox share holders.

The function `_exit()` can be seen in the following code snippet demonstrating the `try-catch` statements and only withdrawing the `available` balance.

```
function _exit() internal override {
  uint256 tokenBalance = aToken.balanceOf(address(this));
  uint256 available = strategyToken.balanceOf(address(aToken));
  if (tokenBalance <= available) {
    // If there are more tokens available than our full position, take all based on aToken balance (continue if unsuccessful).
    try aaveLendingPool.withdraw(address(strategyToken), tokenBalance, address(this)) {} catch {}
  } else {
    // Otherwise redeem all available and take a loss on the missing amount (continue if unsuccessful).
    try aaveLendingPool.withdraw(address(strategyToken), available, address(this)) {} catch {}
  }
}
```

Recommendations

This issue may be mitigated by reverting if the entire balance cannot be withdrawn during an exit. Consider introducing the following code which will revert if the `withdraw()` external call fails.

```
function _exit() internal override {
  // (uint256).max withdraws the entire user balance in Aave
  aaveLendingPool.withdraw(address(strategyToken), (uint256).max), address(this));
}
```

SSBS-05	Non-standard ERC20 Tokens are Not Supported		
Asset	`AaveStrategy.sol`		
Status	**Open**		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

A `strategyToken` which does not strictly follow the ERC20 token interface might not be supported by the contract. Specifically, this is true when the token's `approve()` function does not adhere to the ERC20 interface. One prominent example for such tokens is the stablecoin `USDT`.

When trying to call `_skim()` via `harvest()` or `skim()` of the `BaseStrategy` for such tokens, the transaction reverts. This is because the call to `strategyToken.approve()` in line [**71**] has a different return value of `approve()` on the target contract. In the case of USDT, the function does not return any value, which causes an execution error as a `bool` is expected.

Recommendations

The base contract of `AaveStrategy` defined in `BaseStrategy.sol` is not susceptible to this issue because it's using the "safe" function calls of `SafeTransferLib` for ERC20 tokens (see line [**16**]). The testing team suggests further utilisation of `SafeTransferLib` for `AaveStrategy` and the use of `safeApprove()` and the other functions to interact with the `strategyToken`. Please note, this issue also exists for `SushiStrategy` but, since that contract is meant to operate with the regular ERC20 `SUSHI` token only, the impact is reduced and inconsequential. However, conscious consideration of this issue should be employed when deriving future contracts from `BaseStrategy`.

SSBS-06	Unstaking `stkAave` Extends Cooldown Time		
Asset	`strategies/AaveStrategyMainnet.sol`		
Status	Open		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

When depositing tokens like `USDC` into `Aave Lending Pool` on mainnet, `stkAave` tokens are rewarded to the depositors. Staking `stkAave` should allow relevant parties to unstake and receive `AAVE` tokens as a result. This behaviour requires the `_harvestRewards` function to be called twice:

- The first time when the `stakerCooldowns` is at 0 and needs to be set to the relevant block timestamp.
- The second time to unstake and redeem relevant rewards.

The conditions for unstaking (i.e `cooldown + COOLDOWN_SECONDS < block.timestamp` and `block.timestamp < cooldown + COOLDOWN_SECONDS + UNSTAKE_WINDOW`) are difficult to satisfy. This is as a result of Aave's specific logic, triggered during `incentiveController.claimRewards()`, which updates the `stakerCooldown` through `STAKE_TOKEN.stake()`. Internal logic can then set the `stakerCooldown` to a value that will not meet the requirements of unstaking.

Each time rewards are claimed, the `stakerCooldown` is increased proportionally to the amount of new rewards. Therefore, it will take longer before the rewards can be claimed and redeemed.

Recommendations

The testing team recommends moving `incentiveController.claimRewards()` to only occur outside of the claim window.

```
if (cooldown == 0) {

  incentiveController.claimRewards(rewardTokens, type(uint256).max, address(this));
  stkAave.cooldown();

} else if (cooldown + COOLDOWN_SECONDS < block.timestamp) {

  if (block.timestamp < cooldown + COOLDOWN_SECONDS + UNSTAKE_WINDOW) {

    // We claim any AAVE rewards we have from staking AAVE.
    stkAave.claimRewards(address(this), type(uint256).max);
    // We unstake stkAAVE and receive AAVE tokens.
    // Our cooldown timestamp resets to 0.
    stkAave.redeem(address(this), type(uint256).max);

  } else {

    incentiveController.claimRewards(rewardTokens, type(uint256).max, address(this));

    // We missed the unstake window - we have to reset the cooldown timestamp.
    stkAave.cooldown();

  }
}
```

SSBS-07	Missing Input Validation		
Asset	`StakingContractMainnet.sol`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The contract does not perform validation checks on all user inputs. Making sure that all parameters values fit into the expected value range ensures secure execution under normal operating conditions. Invalid inputs could often lead to unexpected results like lost funds, including increased gas costs. It is also considered good style for writing smart contracts. Apart from the issue described in SSBS-02, there are multiple other occurrences:

- `createIncentive()`
  - `token` should contain bytecode to ensure it is a contract address.
  - `rewardToken` should contain bytecode to ensure it is a contract address.
  - `rewardAmount > 0`, ensures rewards can be distributed.
- `stakeToken()` and `unstakeToken()`
  - `token` should contain bytecode to ensure it is a contract address.
  - `amount > 0`, ensures some tokens are being transferred.
  - For readability, `unstakeToken` may benefit from a `require`, checking that the user has at least `amount` tokens staked. This will currently be caught by an overflow revert on line [216], but no clear error message is provided.
- `subscribeToIncentive()`
  - `0 < incentiveId && incentiveId <= incentiveCount`, as seen in SSBS-02.
  - `userStakes[msg.sender][incentive.token] > 0`, ensures the user has some liquidity staked in the associated token.
- `unsubscribeToIncentive()`
  - `incentiveIndex < userStake.subscribedIncentiveIds.countStoredUint24Values()`, otherwise `userStake.subscribedIncentiveIds.getUint24ValueAt(incentiveIndex)` will return zero.
- `acrueRewards()`
  - `0 < incentiveId && incentiveId <= incentiveCount`, ensures the incentive exists.
- `claimRewards()`
  - `0 < incentiveId && incentiveId <= incentiveCount`, for each incentive ID in `incentiveIds` to ensure these incentives exist.

Recommendations

We recommend adding the aforementioned checks to reduce the impact of user error and prevent malicious misuse.

SSBS-08	`batch()` Incorrectly Interprets Error Messages
Asset	StakingContractMainnet.sol
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `batch()` calls multiple functions in one transaction. If any of the calls fails the transaction will revert and propagate the error message.

There is a bug in the error message propagation which results in an empty string being propagated.

```
(bool success, bytes memory result) = address(this).delegatecall(datas[i]);
if (!success) {
  if (result.length < 68) revert();
  assembly {
    result := add(result, 0x04)
  }
  revert(abi.decode(result, (string)));
}
```

The error occurs in the above code as it assumes the error message in `result` is of type `Error(string)`. The type `Error(string)` occurs when a call reverts via `require(false, "Some error msg")`. The layout of `bytes memory result` will be the first 4 bytes of `keccak256("Error(string)")` followed by the encoding of the `string`. Hence, in the code above it will move the `result` memory pointer forward 4 bytes and then only read the `string` error message.

However, custom errors like those used in `StakingContractMainnet` will have a different layout. The custom errors will have a return type of `keccak256("ErrorName(type1,type2,...)")`. Since the errors in `StakingContractMainnet` do not have additional data, they are all 4 bytes in length.

Hence, when applying the `batch()` error propagation logic, `result` is always 4 bytes for the custom errors and will trigger the `revert` in the line `if (result.length < 68) revert();`.

For example the error `InvalidTimeFrame()` will have `result` equal to the first 4 bytes of `keccak256("InvalidTimeFrame()")`.

Recommendations

Consider wrapping the inner error in a batch error which contains inner results as seen in the following code.

```
error BatchError(bytes innerError);

...
(bool success, bytes memory result) = address(this).delegatecall(datas[i]);

if (!success) {

  revert BatchError(result);

}
```

Resolution

PR #2 will cause the transaction to revert using a custom error to wrap the inner error.

SSBS-09	Using `batch()` to Unsubscribe from Many Incentives Could Lead to Unexpected Results
Asset	`StakingContractMainnet.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Unsubscribing from many incentives using the `batch()` function could lead to unexpected behaviour. In fact, the shift of the incentive index that happens after unsubscribing from an incentive could result in reverting or unsubscribing from a wrong incentive.

The following scenario outlines how the `batch()` might unsubscribe a user from a wrong incentive:

- A user is subscribed to 6 incentives (let's assume that the `incentiveId` is from 1 to 6). All these incentives are for the same token.
- The user decides to unsubscribe from two incentives (e.g `incentiveId` 4 and 6)
- The user decides to unsubscribe from `index 0` (`incentiveId=6`) and `index 2` (`incentiveId=4`) in the `unsubscribeFromincentive()` function to unsubscribe from these two incentives.
- The user uses the `batch()` function to unsubscribe from these two incentives.
- Due to the index shift that happens after unsubscribing from the first incentive, the user finds themselves unsubscribing from incentive `incentiveId=3`.

Recommendations

Make sure this behaviour is understood. The UI should carefully handle this use case. When using the `batch()` function to unsubscribe from many incentives, it should work down from the higher to the lower incentive indices.

Resolution

This issue has been acknowledged by the development team.

SSBS-10	Opportunities for Malicious Tokens to Poison Logs
Asset	`StakingContractMainnet.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Arbitrary `token` contracts are allowed to be staked or given in rewards, this allows malicious actors to poison the logs or revert messages. The development team should be alert to the many strategies used by malicious tokens to perform phishing attacks and other scams.

This contract is vulnerable to a misleading logs and revert message. The attack on line [**228**] of `unstakeToken()` could interact with a malicious token to create the illusion of trapped tokens. A user can set `transferExistingRewards` to `false` to withdraw without interacting with potentially malicious reward tokens but, if they attempt to withdraw all of their stake, the statement on line [**228**] will cause all incentives to be claimed. If the user is subscribed to a malicious reward, the revert will prevent any of their tokens from being returned and they may believe they are locked in.

There are also attacks based on exploiting `tx.origin` in other contracts which could be exploitable by offering a bogus reward token in an incentive that would attempt attacks when claimed.

Recommendations

These issues mainly rely on misleading the user into harmful actions, and they are not easily preventable in the contract code. Nevertheless, if all tokens are allowed, the team should remain alert to token scam strategies and take steps to prevent them in UI design.

In particular, a user withdrawing all of their stake needs to be aware that, if the transaction fails, they can unsubscribe and claim from individual incentives to fix the issue.

Resolution

This is a known issue in the EVM that external calls may produce malicious logs and revert messages. The development team have acknowledged this as a potential threat.

SSBS-11	Contracts Do Not Implement Safe Ownership Transfer Pattern
Asset	`Ownable.sol`
Status	**Open**
Rating	Informational

Description

The current transfer of ownership pattern calls the function `transferOwnership(address newOwner)` which instantly changes the owner to the `newOwner`. This allows the current owner of the contracts to set an arbitrary address (excluding the zero address).

If the address is entered incorrectly or set to an unowned address, the owner role of the contract is lost forever. Thus, a user would not be able to pass the `onlyOwner` modifier.

Similarly, the function `renounceOwnership()` allows an owner to remove themself as owner and prevent any future owners. Again, this will cause any function using the `onlyOwner` modifiers to always fail.

Recommendations

This scenario is typically mitigated by implementing a two-step `transferOwnership` pattern, whereby a new owner address is selected, then the selected address must call a `claimOwnership()` before the owner is changed. This ensures the new owner address is accessible.

SSBS-12	Miscellaneous `BentoBox Strategies and Staking Contract` General Comments
Asset	`bentobox-strategies` and `staking-contract`
Status	Open
Rating	Informational

Description

This section details miscellaneous findings in the `bentobox-strategies` and `staking-contract` repository that do not have direct security implications:

1. **`bentobox-strategies/BaseStrategy.sol`**

   1a) No revert message for the require statement in `afterExit()`
   The require statement in line [252] just checks for the bool value without a revert message. Consider adding a revert message.

   1b) Typo
   line [59]: *"whith"*, should be *"with"*
   line [227]: *"locked if its (accidentally) set twice in"*, should be *"it's"*

   1c) Unused internal function
   There is an unused internal function `increment()` that can be removed.

   1d) Public functions
   `maxBentoBoxBalance()` and `exited()` are `public` functions that can be made `external` as they are not called internally in the contract.

   1e) There are currently no liquidity mining rewards active on `AAVE v2` on `Polygon`.

2. **`bentobox-strategies/helpers/Harvester.sol`**

   2a) Typo
   line [80]: function name `_rebalanceNecessairy`, should be `_rebalanceNecessary`

3. **`staking-contract/StakingContractMainnet.sol`**

   3a) Remove unused import `test/Console.sol`.

   3b) Gas Optimisations
   - Memory variables can be assigned to `incentive.lastRewardTime` and `incentive.endTime` in `_accrueRewards()` to avoid extra `SLOAD` operations.
   - Memory variable can be assigned to `rewardPerLiquidityLast[msg.sender][incentiveId]` in `_calculateReward` to avoid extra `SLOAD` operations.

   3c) Typo
   line [45]: *"vlaues"*, should be *"values"*

   3d) Restrict function visibility
   The testing team recommends adhering to the strictest visibility possible on all functions. As a result, `_calculateReward` can be restricted to `view` visibility as the function does not write to storage.

   3e) Misleading event emission
   The event emission in `updateIncentive` in line [149] is executed even if no incentive parameters have changed. This could affect off-chain monitoring applications.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

Staking Contract Tests:

```
test_createIncentive                                  PASSED  [3%]
test_createIncentive_early_start_time                 PASSED  [6%]
test_createIncentive_invalid_reward_token             PASSED  [10%]
test_createIncentive_invalid_time_frame               PASSED  [13%]
test_stakeToken_no_subscriptions                      PASSED  [16%]
test_stakeToken_with_subscriptions                    PASSED  [20%]
test_stakeToken_with_rewards                          PASSED  [23%]
test_unstakeToken_save_rewards                        PASSED  [26%]
test_unstakeToken_transfer_rewards                    PASSED  [30%]
test_subscribeToIncentive                             PASSED  [33%]
test_subscribeToIncentive_fully_subscribe             PASSED  [36%]
test_subscribeToIncentive_already_subscribe           PASSED  [40%]
test_subscribeToIncentive_which_dont_exist            PASSED  [43%]
test_stakeAndSubscribeToIncentives                    PASSED  [46%]
test_stakeAndSubscribeToIncentives_id_zero            PASSED  [50%]
test_stakeAndSubscribeToIncentives_repeate_incentive  PASSED  [53%]
test_unsubscribeFromIncentive                         PASSED  [56%]
test_unsubscribeFromIncentive_resubscribing           PASSED  [60%]
test_unsubscribeFromIncentive_not_subscribed          PASSED  [63%]
test_claimRewards_repeating_incentive                 PASSED  [66%]
test_claimRewards_not_subscribed                      PASSED  [70%]
test_claimRewards_before_start                        PASSED  [73%]
test_updateIncentive                                  PASSED  [76%]
test_updateIncentive_start_time_early                 PASSED  [80%]
test_updateIncentive_invalid_time_frame               PASSED  [83%]
test_updateIncentive_min_value                        PASSED  [86%]
test_batch_error_messages                             PASSED  [90%]
test_batch                                            PASSED  [93%]
test_attackToken                                      PASSED  [96%]
test_badToken                                         PASSED  [100%]
```

BentoBox Strategy Tests:

```
test_SafeHarvest                                      PASSED  [5%]
test_harvest_withdraw                                 PASSED  [10%]
test_skim                                             PASSED  [15%]
test_targetPercentage                                 PASSED  [20%]
test_setStrategyExecutor                              PASSED  [25%]
test_exit                                             PASSED  [30%]
test_afterExit_not_owner                              PASSED  [35%]
test_afterExit_before_exit                            PASSED  [40%]
test_afterExit                                        PASSED  [45%]
test_setSwapPath                                      PASSED  [50%]
test_setSwapPath_with_strategy_token_as_in_token      PASSED  [55%]
test_setSwapPath_not_owner                            PASSED  [60%]
test_swapExactTokens                                  PASSED  [65%]
test_exit_not_bento_box                               PASSED  [70%]
test_exit_through_setstrategy                         PASSED  [75%]
test_targetPercentage                                 PASSED  [80%]
test_skim                                             PASSED  [85%]
test_harvest_withdraw                                 PASSED  [90%]
test_withdraw                                         PASSED  [95%]
test_exit                                             PASSED  [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

σ'