

# Final Audit: Nali Finance



## Brewlabs Services Hub

Summary: Nali Finance is a BEP20 project with the vision to improve supply chain processes using blockchain. It is written in Solidity and compiled using build 0.8. The contract ecosystem supports staking and NFT technology.

**Disclaimer:** This audit documentation is for discussion purposes only. The scope of this audit was to analyze and document Nali smart contract codebase for quality, security, and correctness. This audit guarantees that your code has been revised by an expert.

**Audit package:** Standard security, with logic check

**Deployment Status:** Unknown

**Files Audited:** NaliToken.sol, NaliSwap.sol, NaliProduct.sol, NaliNFTTicket.sol, NaliNFTFinancial.sol, Nali1155.sol, Products.sol. Standard libraries such as SafeMath and ERC20 were not audited.

**Audit Result:** **Security PASSED, Logic PASSED**

## Issue Categorisation

Security		Logic	
●●	Critical - Requires immediate review by Development Team	●	Critical - Requires immediate review by Development Team.
●●	Major - Review strongly recommended	●	Major - Review strongly recommended
●●	Minor - Review recommended	●	Minor - Review recommended
●●	Informational - Optional implementation	●	Informational - Optional implementation

This audit is based on the latest versions of the contract files made available to Brewlabs by the Nali Development Team. It has been carried out after the Preliminary Notes (Appendix A) were reviewed by the Nali Development team and subsequent to any updates to contract files as a result of the review.

## Overview of Audit

0 critical security issues identified.

0 critical logic issue identified.

2 major security issue identified.

0 major logic issues identified.

Some minor and informational issues identified across security and logic.

## General Issues

- Owners can alter contract fees. Lack of input validation on fee/price declarations and updates poses centralisation risk. No validation or condition handling to encourage owners to input a reasonable value and prevent unethical behaviour, such as setting new fees to 100 or 0.

a. *NaliSwap*: Line 83 ( *\_swapRate* )

*Response from Development Team:* Limited to 300 (30%) for the *swapFee*. An upper limit for *\_swapRate* cannot be set because it can be anything more than zero, it's like if an exchange would limit its rate on BTC to 1:100000\$ limiting the max price of the asset to 100k.

b. *NaliProduct*: Line 152 ( *quickSellPercentage* )

*Response from Development Team:* *quickSellPercentage* could span from 0 to 100% but it's the value which the user can earn (only with secondary product) selling their position instead of selling the NFT Financial tied to it's position. It's a calculated value that is set at the beginning, usually 50% but connected to the % of secondary product buyers the more secondary product buyers the more *quickSellPercentage*.

2. ● Explicit declaration of False Booleans and zero uint for variables not required since false and zero are defaults in Solidity, respectively. Recommendation is to declare the variable as: `bool {visibility} {variable_name};` or `uint {visibility} {variable_name};` to improve gas usage.
    - a. *NaliProduct*: Line 279
    - b. *NaliFinancial*: Line 11
    - c. *NaliNFTTicket*: Line 21
  3. ● Moderate use of comments across all files. Comments help to improve readability and is encouraged. Refer to the [NatSpec Solidity](#) guidelines for further information.
  4. ● Layout of files does not always follow [Solidity Guidelines](#). Please check the guidelines and adjust as necessary.
- 

## 2. Contract Specific Issues:

### a. *NaliSwap*

1. ●● Centralisation risk for community with contract allowing owners to transfer all BNB accrued in the contract to an onlyOwners wallet (Line 262). This issue violates the ethics of Decentralised Finance. Recommendation is to mitigate this risk by ensure the Development team is doxxed or consider Know Your Customer (KYC) services. *Response from Development Team:* The `withdraw()` function is mandatory and does not represent risk of centralization, NaliSwap swaps ERC20 Nali <-> ERC1155, we may apply a fee for a swap, withdraw allow us to withdraw the collected fees in contract.

### b. *NaliProduct*

1. ●● Centralisation risk for community with contract allowing owners to transfer all BNB accrued in the contract to an onlyOwners wallet (Line 489). This issue violates the ethics of Decentralised Finance. Recommendation is to mitigate this risk by ensure the Development team is doxxed or consider Know Your Customer (KYC) services. *Response from Development Team:* Require a way to withdraw BNB from the contracts, in this case it's the core of the NaliProduct, which allows us to withdraw BNB to start the products. The standard usage is one of our contracts and/or our server wallet withdraw BNB to swap in other currencies, bridge them and buy assets or NFTs or what's required from product. Team have added the amount parameter to allow a more granular withdraw from the contract to help alleviate this risk.
  2. ● Development team to confirm if the status of the secondary product should always be consistent with status of the primary product. If this is the case, there are several instances across this contract where it is possible for different statuses to occur. For example:
    1. In the `finalize()` function, it could be possible for:
      - a. The status of the secondary product to be set to `NOT_ACTIVATED` if the block timestamp is greater than its `endDate` and its `endDate` is < `endDate` of the primary product (Line 141)
      - b. The status of the secondary product to remain unchanged when the status of the primary product is updated to `MINTING` (Line 136)
      - c. The status of the secondary product to remain unchanged when the status of the primary product is updated to `PRE-VOTE` (Line 130)
  3. ● Lack of input validation to check if the balance in the contract exceeds or is equal to the *amount* being withdrawn in `withdraw()` function (Line 489). Recommendation is to add a `require` statement to check if the balance is greater than or equal to the *amount* being withdrawn.
  4. ● In the `getByIds()` function it would be more efficient to iterate over the *ids* input array rather than indirectly iterating via the `_productMaxId`, particularly for high values of `_productMaxId`.
-

Clarification required from the Development team regarding NaliProduct.sol functionality:

1. `finalize()` function is public and can be called by anyone. Development team to confirm design intention.

*Response from Development Team:* Intended behaviour, no change required.

2. Currently all buyers are refunded their purchases of a product when a downvote is received by any user for that product (Line 203). Assumed behaviour would have been that a user who is a purchaser can only downvote, and secondly, if a downvote is received, then only the refund for the purchase made by the downvote is issued. Development team to confirm design intention.

*Response from Development Team:* Intended behaviour, no change required - the product is immediately active if all buyers vote True, otherwise if any vote False, then the contract refunds all buyers including those of the secondary product id.

3. Currently a product proceeds to NOT ACTIVATED when the endDate elapses and not all freeSlots are bought. Assumed behaviour was that the product proceeds to PRE-VOTE if at least some freeSlots were bought by users. Development team to confirm design intention.

*Response from Development Team:* Intended behaviour, no change required - PRE-VOTE status is only applicable to products without a secondary id. Products with secondary id with freeSlots > 0 go in PRE-VOTE.

4. Currently `distributeAndMint` can occur when the product is in any status. Assumed behaviour was that this may only be applicable when the status is MINTING. Development team to confirm design intention.

*Response from Development Team:* Intended behaviour, no change required - distribution can occur in more than one state: CREATED, ACTIVE, VOTING and MINTING. It's callable only from owners and/or Nali contracts.



**Conclusion:** The Brewlabs team thank you for the opportunity to review and audit your smart contract code. The data from this report will be formalised in the audit publication for your community. Keep in touch as we offer discounts for repeat business and on a range of other services!

The Brewlabs Team, 2022-04-26

---

## Preliminary Notes: Nali Finance



### Brewlabs Services Hub

Summary: Nali Finance is a BEP20 project with the vision to improve supply chain processes using blockchain. It is written in Solidity and compiled using build 0.8. The contract ecosystem supports staking and NFT technology.

**Disclaimer:** This audit documentation is for discussion purposes only. The scope of this audit was to analyze and document Nali smart contract codebase for quality, security, and correctness. This audit guarantees that your code has been revised by an expert.

**Audit package:** Standard security, with logic check

**Deployment Status:** Unknown

**Files Audited:** NaliToken.sol, NaliSwap.sol, NaliProduct.sol, NaliNFTTicket.sol, NaliNFTFinancial.sol, Nali1155.sol, Products.sol. Standard libraries such as SafeMath and ERC20 were not audited.

**Audit Result:** **Security FAILED, Logic FAILED**

### Issue Categorisation

Security		Logic	
●●	Critical - Requires immediate review by Development Team	●	Critical - Requires immediate review by Development Team.
●●	Major - Review strongly recommended	●	Major - Review strongly recommended
●●	Minor - Review recommended	●	Minor - Review recommended
●●	Informational - Optional implementation	●	Informational - Optional implementation

### Overview of Audit

1 critical security issues identified.

1 critical logic issue identified.

3 major security issue identified.

5 major logic issues identified.

Several minor and informational issues identified across security and logic.

### 1. General Issues

- Missing events for changes to state variables in various functions. This means only the contract owner can alter state variables without the community being aware of changes. Recommendation is to add events for all changes to state variables or communicate to community when relevant changes are committed to improve transparency
  - NaliSwap: Line 58, 62, 67, 71, 76
  - NaliProduct: Line 51, 55, 59, 63, 67, 411
  - NaliNFTTicket: Line 78, 90, 94, 102
  - NaliNFTFinancial: Line 30
- Lack of input validation on fee/price declarations and updates poses centralisation risk. No validation or condition handling to encourage owners to input a reasonable value and prevent unethical behaviour, such as setting new fees to 100 or 0.
  - NaliSwap: Line 71, 76 (`_swapFee`, `_swapRate`)
  - NaliProduct: Line 143 (`quickSellPercentage`)
  - NaliNFTTicket: Line 82 (`_price`)
- Lack of input validation for wallet and contract address changes which could lead to unexpected loss if an invalid address is set, for example, `address(0)`.

- a. *NaliSwap*: Line 58, 62, 67
  - b. *NaliProduct*: Line 51, 55, 63
  - d. *NaliNFTTicket*: Line 94
4. ● Visibility not explicitly set for state variables. By default the visibility is internal. Development team to confirm if this is the correct visibility.
- a. *NaliSwap*: Line 19, 20
5. ● State variables could be set to constant as the variables are never modified.
- a. *NaliToken*: Line 7, 8
  - b. *NaliSwap*: Line 24, 28
6. ● Function visibility is set to *public*, but are never called by the contract. Recommendation is to set the function visibility to *external* to improve gas efficiency.
- a. *NaliSwap*: Line 67, 71, 76, 92
  - b. *NaliProduct*: Line 51, 55, 59, 63, 67, 72
  - c. *NaliNFTTicket*: Line 82,
7. ● Explicit declaration of False Booleans and zero uint for variables not required since false and zero are defaults in Solidity, respectively. Recommendation is to declare the variable as: *bool {visibility} {variable\_name};* or *uint {visibility} {variable\_name};* to improve gas usage.
- a. *NaliProduct*: 362, 364, 457,
  - b. *NaliNFTTicket*: Line 35, 127
8. ● Naming of state variables representing contracts do not follow Solidity style guidelines
- a. *NaliSwap*: Line 21, 22
  - b. *NaliProduct*: Line 16, 17, 18
  - c. *NaliNFTTicket*: Line 20
9. ● Moderate use of comments across all files. Comments help to improve readability and is encouraged. Refer to the [NatSpec Solidity](#) guidelines for further information.
10. ● Layout of files does not always follow [Solidity Guidelines](#). Please check the guidelines and adjust as necessary.
11. ● SafeMath library is obsolete for [Solidity Builds >= 0.8.0](#), with arithmetic operations automatically reverting on underflow and overflows.
- 

## 2. Contract Specific Issues:

### b. *NaliSwap*

1. ●● Centralisation risk for community with contract allowing owners to transfer all BNB accrued in the contract to an onlyOwners wallet (Line 255). This issue violates the ethics of Decentralised Finance. Recommendation is to mitigate this risk by ensure the Development team is doxxed or consider Know Your Customer (KYC) services
2. ● *\_swapFee* can be set zero despite message in require statement stating it cannot be zero (Line 72). Development team to confirm if the comparison sign should be greater than or equal to, or greater than.
3. ● *\_factory* variable is declared but never used. Recommendation is to remove unused variables
4. ● Spelling errors Line 147, 176.

## b. NaliProduct

1. ●● Violation of Check-Effects-Interaction pattern with state changes made to buyers shares after calls to an external payable address are made (Line 180, 181) in the `_refund()` function. The contract sets the value of the buyers share (`_products[productId].purchases[k].shares`) after transfers are made to an external address (contract or wallet) which can potentially lead to reentrancy attacks. Recommendation is to make all state changes prior to engaging external contracts/wallets and use the `noReentrant` modifier
2. ● In the `changeProductData()` function, a `secondaryId` can be set to `true` for a product, however the `isSecondary` attribute of the secondary product is not updated as part of the function. Initial recommendation is to remove the functionality to update the `secondaryId` from the `changeProductData` function so that updates are redirected through the `setSecondaryId()` function. If that is not viable, secondary recommendation is to expand the if statement to include the following line (highlighter green):

```
if (setSid) {  
    _products[id].secondaryId = secondaryId;  
    _products[secondaryId].isSecondary = true;  
}
```

3. ●● Centralisation risk for community with contract allowing owners to transfer all BNB accrued in the contract to an onlyOwners wallet (Line 484). This issue violates the ethics of Decentralised Finance. Recommendation is to mitigate this risk by ensure the Development team is doxxed or consider Know Your Customer (KYC) services
4. ● When setting a new `_defaultFee`, the `newValue` must be greater than 100 for the change to proceed. This is higher than the default value of the `_defaultFee` (70). Development team to confirm if the comparison sign should be less 100, or whether it should remain as greater than 100.
5. ● Lack of validation checks in the `activate()` and `flag()` function to ensure the both `endDate` and `startDate` > 0 and `endDate` > `startDate`. Recommendation is to add basic input validation checks to ensure that code operates as expected.
6. ● Lack of validation checks to ensure the secondary product exists before being set for a product. Recommendation is to add a require statement to the `setSecondaryId()` function that checks if the `id` of the secondary product is equal to the input `sId`. For example:  

```
require(_products[sId].id == sId, "Secondary product does not exist");
```
7. ● Lack of validation checks to ensure the uint passed in for the `productStatus` in the `flag()` function is within the range of the allowable index range of the `ProductStatus` enum. Recommendation is to add a require statement to ensure passed uint is in permitted range.
8. ● In the `activate()` function, `votes` is declared as an array of booleans with `spots` used to determine the length of the array, however `spots` could be zero if `tickets` is passed in as zero. The value of `tickets` or `freeslots` for a product cannot be updated once activated. Development team to confirm if a require statement is required to ensure `tickets` > 0, otherwise users will not be able to buy or vote on the product (prevented by the require statement on Line 312), and therefore the creation of the product appears redundant.
9. ● No functionality to unset a `secondaryId` for a product. Expected functionality would be in the `changeProductData` function when `setSid` is passed in as false. Development team to confirm if this is the intended functionality.
10. ● Development team to confirm if the status of the secondary product should always be consistent with status of the primary product. If this is the case, there are several instances across this contract where it is possible for different statuses to occur. For example:
  1. The status of the secondary product is not updated with the status of the primary product as part of the `setSecondaryId()` function

2. In the *finalize()* function, it could be possible for:

- a. The status of the secondary product to be set to *NOT\_ACTIVATED* if the block timestamp is greater than its *endDate* and its *endDate* is < *endDate* of the primary product (Line 133)
- b. The status of the secondary product to remain unchanged when the status of the primary product is updated to *MINTING* (Line 128)
- c. The status of the secondary product to remain unchanged when the status of the primary product is updated to *PRE-VOTE* (Line 121)

11. ● Commented out code could be removed if not required Lines 113, 127, 129, 215 - 230, 308

12. ● In the *getByIds()* function it would be more efficient to iterate over the *ids* input array rather than indirectly iterating via the *\_productMaxId*, particularly for high values of *\_productMaxId*.

13. ● The *votes* array is not updated when a downvote is received as part of the *vote()* function. Development team to confirm if this functionality is required.

#### c. *NaliNFTTicket*

1. ● ● Violation of Check-Effects-Interaction pattern with state changes made to users claimable tickets after calls to an external payable address are made (Line 120, 121) in the *claimTicketRewards()* function. The contract resets the value of *claimableTicket[msg.sender]* after transfers are made to an external address (contract or wallet) which can potentially lead to reentrancy attacks. Recommendation is to make all state changes prior to engaging external contracts/wallets, in addition to the re-entrancy modifier (already applied to this function).

---

Clarification required from the Development team regarding *NaliProduct.sol* functionality:

1. *finalize()* function is public and can be called by anyone. Development team to confirm design intention.
2. Currently all buyers are refunded their purchases of a product when a downvote is received by any user for that product (Line 190). Assumed behaviour would have been that a user who is a purchaser can only downvote, and secondly, if a downvote is received, then only the refund for the purchase made by the downvote is issued. Development team to confirm design intention.
3. Currently a product proceeds to *NOT\_ACTIVATED* when the *endDate* elapses and not all *freeSlots* are bought. Assumed behaviour was that the product proceeds to *PRE-VOTE* if at least some *freeSlots* were bought by users. Development team to confirm design intention.
4. Currently *distributeAndMint* can occur when the product is in any status. Assumed behaviour was that this may only be applicable when the status is *MINTING*. Development team to confirm design intention.



**Conclusion:** The Brewlabs team thank you for the opportunity to review and audit your smart contract code. The data from this report will be formalised in the audit publication for your community. Keep in touch as we offer discounts for repeat business and on a range of other services!

The Brewlabs Team, 2022-04-15