#HashEx
BLOCKCHAIN SECURITY

Zunami Native

smart contracts final audit report

May 2023

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Zunami Protocol team to perform an audit of their smart contract. The audit was conducted between 26/04/2023 and 01/05/2023.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at @ZunamiProtocol/ZunamiProtocol GitHub repo after the 0d037e94 commit.

Update. A recheck was done after the commit 7568c0f.

2.1 Summary

Project name	Zunami Native
URL	https://www.zunami.io/
Platform	Ethereum
Language	Solidity

2.2 Contracts

Name	Address
ZunamiNative	
StakingEthFraxCurveConvexStratBase	
SellingCurveRewardManagerNative	
ZunamiRebalancer	

3. Found issues

Critical	1 (5%)
High	3 (15%)
Medium	3 (15%)
Low	10 (50%)
Info	3 (15%)

C1. ZunamiNative

ID	Severity	Title	Status
C1-01	Critical	Reentrancy	Resolved
C1-02	High	Owner (admin) exaggerated rights	Acknowledged
C1-03	High	Default deposit/withdraw pid change	Acknowledged
C1-04	Medium	Move funds without native tokens	Resolved
C1-05	Low	Gas optimization	Partially fixed
C1-06	Low	Lack of events	Resolved
C1-07	Low	Parameters validation	Resolved
C1-08	Low	Implicit default tokens[0] value	Resolved
C1-09	Low	Depositing more than 3 tokens	Acknowledged
C1-10	Info	Incomplete NatSpec documentation	Partially fixed
C1-11	Info	Pauser role	Resolved

C1-12	Info	Code consistency	Resolved

C2. StakingEthFraxCurveConvexStratBase

ID	Severity	Title	Status
C2-01	High	Exaggerated owner rights	Acknowledged
C2-02	Medium	Unused branching in convertZunamiTokenIdToPoolOne()	Resolved
C2-03	Medium	Curve's reentrancy	Acknowledged
C2-04	Low	Lack of constructor parameters validation	Acknowledged
C2-05	Low	Gas optimization	Partially fixed

C3. SellingCurveRewardManagerNative

ID	Severity	Title	Status
C3-01	Low	Hardcoded decimals	Acknowledged
C3-02	Low	Gas optimization	Resolved
C3-03	Low	Ingnored oracle update timestamp	Resolved

4. Contracts

C1. ZunamiNative

Overview

The contract is an ERC20 standard implementation. It provides an entrypoint for users willing to invest in the strategies. One of the investing tokens is native EVM currency.

After the deposit, a user receives Zunami LP (ZLP) tokens reflecting his share in total holdings across all strategies. ZLP tokens can be exchanged for any of the strategies' underlying assets using one of withdraw functions. It supports gas-optimized deposit/withdraw methods via delegation to the operator, who executes a batch of multiple users' requests.

Issues

C1-01 Reentrancy

Critical Resolved

The `removePendingDeposit()` function allows users to remove their funds from the contract before they are invested in the strategy.

Deletion of user records about deposits occurs after calling the `safeTransferNative()` function. Thus, an attacker can take over control when transferring native tokens to a controlled contract in the `safeTransferNative()` function. This will allow to re-enter the `removePendingDeposit()` function to drain all pending deposits.

```
    function removePendingDeposit() external {
        uint256 pendingDeposit_;
        for (uint256 i = 0; i < POOL_ASSETS; i++) {
            pendingDeposit_ = _pendingDeposits[_msgSender()][i];
            if (pendingDeposit_ > 0) {
                safeTransferNative(tokens[i], _msgSender(), pendingDeposit_);
            }
        }
```

```
        delete _pendingDeposits[_msgSender()];
        emit RemovedPendingDeposit(_msgSender());
    }

    function safeTransferNative(
        address token,
        address receiver,
        uint256 amount
    ) internal {
        if (address(token) == ETH_MOCK_ADDRESS) {
            receiver.call{ value: amount }('');
        } else {
            IERC20Metadata(token).safeTransfer(receiver, amount);
        }
    }
```

Recommendation

1. Deletion of user's `_pendingDeposits` data must be completed before transfers processing.

2. We strongly recommend using the reentrance guard pattern in all functions where native tokens are transferred.

C1-02 Owner (admin) exaggerated rights

High Acknowledged

The contract admin can execute the `withdrawStuckToken()` function to withdraw any token from the contract. It also allows to withdraw tokens that have been deposited by users using the `delegateDeposit()` function (`_pendingDeposits`).

Recommendation

We recommend restricting the admin's ability to withdraw the tokens used for the deposit.

Team response

Admin role will be switched to DAO.

C1-03 Default deposit/withdraw pid change

High Acknowledged

a. The admin has to control the actual withdrawal pid has a sufficient `lpshares` amount to service users withdraw needs, otherwise, users will be unable to return their assets;

b. The admin has to make sure while setting default withdraw/deposit pool, that addressed strategy has a similar or close price of used liquidity token for staking, otherwise it will cause unfair distribution of `lpshares`.

```
function setDefaultDepositPid(uint256 _newPoolId)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
    enabledPool(_newPoolId)
{
    require(_newPoolId < _poolInfo.length, 'wrong pid');
    defaultDepositPid = _newPoolId;
    emit SetDefaultDepositPid(_newPoolId);
}

function setDefaultWithdrawPid(uint256 _newPoolId)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
    enabledPool(_newPoolId)
{
    require(_newPoolId < _poolInfo.length, 'incorrect pid');
    defaultWithdrawPid = _newPoolId;
    emit SetDefaultWithdrawPid(_newPoolId);
}
```

Recommendation

Consider reworking the mechanism of changing active PID to allow users to choose individually or update the default one if the admin is not responding.

Team response

Admin role will be switched to DAO.

C1-04 Move funds without native tokens

Medium Resolved

The function `moveFundsBatch()` allows transferring funds to another strategy.

But the balance of the native `ETH_MOCK_ADDRESS` token isn't properly handled in the `moveFundsBatch()` function.

```
function moveFundsBatch(
    uint256[] memory _strategies,
    uint256[] memory withdrawalsPercents,
    uint256 _receiverStrategy
) external onlyRole(DEFAULT_ADMIN_ROLE) enabledPool(_receiverStrategy) {
    ...
    for (uint256 y = 0; y < POOL_ASSETS; y++) {
        address token = tokens[y];
        if (token == address(0)) break;
        tokenBalance[y] = IERC20Metadata(token).balanceOf(address(this));
    }
    ...
}
```

Recommendation

Make sure the function works as intended, otherwise, the token balance handling needs to be added to the function.

C1-05 Gas optimization

Low Partially fixed

a. The state variable `ETH_MOCK_ADDRESS` can be declared as constant to save gas.

b. The state variable `tokenCount` can be cast to uint128 type (or lower) to be stored in the one storage slot with the `availableWithdrawalTypes` variable. It can be done by moving it to L60.

c. Multiple reads of the `tokenCount` variable from storage in the `addTokens()` function. Consider using a local variable.

d. Multiple reads of the `defaultDepositPid` variable from storage in the `processSuccessfulDeposit()`, `withdraw()`, `processSuccessfulWithdrawal()`, `completeWithdrawalsBase()` function. Consider using a local variable.

e. Unchecked math could be used in the all `for()` loops for `++i` counter as well as other counters with fixed increments.

f. Instead of multiple storage readings, the local variable can be used for the `_poolInfo.length` value in the `autoCompoundAll()`, `totalHoldings()`, `addPool()` functions.

g. The local variable `withdrawnTokens` of the `completeWithdrawalsOneCoin()` function is never used. Check that the variable is really not needed and remove it.

h. The whole `PoolInfo` structure is read in the `autoCompoundAll()` and `totalHoldings()` functions, but the startTime field is not used.

C1-06 Lack of events

Low Resolved

We recommended emitting events on important value changes to be easily tracked off-chain.

No event is emitted in the `setAvailableWithdrawalTypes()` function.

C1-07 Parameters validation

Low Resolved

1. Consider adding validation for the input array lengths of the `addTokens()` function.

2. Consider adding validation for the input parameters of the `addTokens()`, `replaceToken()` functions to check decimals values and non-zero checks for token addresses.

C1-08 Implicit default tokens[0] value

Low Resolved

The `deposit()` function implicitly defines `tokens[0] == ETH_MOCK_ADDRESS`, which is not true in general.

Consider executing the `addTokens()` function for the `ETH_MOCK_ADDRESS` in the contract constructor.

C1-09 Depositing more than 3 tokens

Low Acknowledged

The `deposit()`, `delegateDeposit()` functions allow users to deposit up to 5 different tokens. All these tokens will be transferred to the StakingEthFraxCurveConvexStratBase contract. At the same time, the StakingEthFraxCurveConvexStratBase contract allows the processing of no more than 3 tokens.

Thus, if the StakingEthFraxCurveConvexStratBase contract receives more than 3 tokens, they will be locked in it. Basically, unlocking such tokens is possible using the `withdrawStuckToken()` function. But it can be very time-consuming and costly in terms of gas.

Recommendation

Consider using the same values for the array length of the `tokens` (ZunamiNative contract) variable and value of the `STRATEGY_ASSETS` variable (StakingEthFraxCurveConvexStratBase contract).

C1-10 Incomplete NatSpec documentation

Info Partially fixed

1. Some functions of the contract do not have documentation, or it is incomplete.We recommend writing documentation using NatSpec Format. This would help in development, as well as simplify user interaction with the contract (including using the block explorer).

2. The `setManagementFee()` function has an incorrect description for the `newManagementFee` parameter (about maxAmount condition).

C1-11 Pauser role

Info Resolved

We recommend introducing a separate `PAUSER` role to implement monitoring without risk of compromising the `ADMIN` key.

C1-12 Code consistency

Info Resolved

For code consistency in the `getWithdrawalSafe()` function, we recommend explicitly specifying the return value of the `IStrategy.WithdrawalType` variable, depending on the input parameter `neededType`.

C2. StakingEthFraxCurveConvexStratBase

Overview

A strategy contract to supply incoming funds into native Curve ETH/fraxETH pool and staking received LPs to a Curve staking vault. All rewards can be compounded, funds can be withdrawn in base assets or in a single coin.

Issues

C2-01 Exaggerated owner rights

High Acknowledged

The `setZunami()` function allows the owner to update the Zunami contract address, which has access to withdrawals. In other words, the owner has full access to all strategies funds by setting the `zunami` address to EOA or malicious contract.

```
    function setZunami(address zunamiAddr) external onlyOwner {
        zunami = IZunami(zunamiAddr);
    }
```

The `setRewardManager()` function also grants the contract's owner access to set malicious siphoning contracts as RewardManager.

Recommendation

Consider making these functions initializers, i.e. being callable only once.

Another way is to renounce the ownership or transfer it to a Timelock-like contract with MultiSig admin.

Team response

Admin role will be switched to DAO.

C2-02 Unused branching in convertZunamiTokenIdToPoolOne()

Medium Resolved

The `convertZunamiTokenIdToPoolOne()` function returns only `iETH_frxETH_POOL_ETH_ID` value in all cases.

```
    function convertZunamiTokenIdToPoolOne(uint256 zunamiTokenId) internal pure returns
(int128) {
        if (zunamiTokenId == ZUNAMI_ETH_TOKEN_ID || zunamiTokenId == ZUNAMI_wETH_TOKEN_ID)
{
```

```
            return iETH_frxETH_POOL_ETH_ID;
        } else {
            return iETH_frxETH_POOL_ETH_ID;
        }
    }
```

Recommendation

Use `iETH_frxETH_POOL_frxETH_ID` value for else cases.

C2-03 Curve's reentrancy

Medium Acknowledged

Curve's `get_virtual_price()` is susceptible to reentrancy manipulation, any external functions should be protected against reentrancy.

```
    function getCurvePoolPrice() internal view returns (uint256) {
        return fraxEthPool.get_virtual_price();
    }
```

While the `deposit()` and `withdraw()` functions of the StakingEthFraxCurveConvexStratBase contract don't suffer this issue directly, the result of `totalHoldings()` can be manipulated.

Recommendation

To mitigate risks, in all functions that use the `get_virtual_price()` value, we suggest calling one of the Curve pool functions that have the `@nonreentrant('lock')` modifier. The arguments to the called function can be null values (e.g. `remove_liquidity(0, [0, 0])`).

Triggering the `nonreentrant` modifier will prevent the `get_virtual_price()` function response from being manipulated.

C2-04 Lack of constructor parameters validation

Low Acknowledged

Consider adding validation for input constructor parameters, to prevent incorrect contract initialization.

C2-05 Gas optimization

Low Partially fixed

a. The state variable `ETH_MOCK_ADDRESS` can be declared as constant to save gas.

b. The `autoCompound()`, `updateMinDepositAmount()`, `claimManagementFees()`, `totalHoldings()` functions can be declared as `external` to save gas.

c. The internal function `wrapETH()` is never used and can be removed.

d. Multiple reads of the `kekId` variable from storage in the `stakeCurveLp()` function. Consider using a local variable.

e. Multiple reads of the `_config.rewards[i]` from storage in the `sellRewards()` function. Consider using a local variable.

f. The variables `feeTokenId`, `_config.tokens[feeTokenId_].balanceOf(address(this))`, and `managementFees` in the `autoCompound()` function can be obtained from the `sellRewards()` function without storage reading.

C3. SellingCurveRewardManagerNative

Overview

The contract serves for swapping reward tokens for native tokens (Ether) with a 3% of slippage allowance.

Issues

C3-01 Hardcoded decimals

Low Acknowledged

Price-feeds decimals are implicitly hardcoded into the `checkSlippage()` function.

Consider using explicit assignment of the price-feeds decimals in the contract constructor.

C3-02 Gas optimization

Low Resolved

a. The private variable `feeCollector` is never used and can be removed.

b. The `hande()` function can be declared as `external` to save gas.

C3-03 Ingnored oracle update timestamp

Low Resolved

Price-feed's `updatedAt` timestamp value is ignored in the `checkSlippage()` function.

Consider adding validation for the timestamp to ensure that the price feed data is not stale.

C4. ZunamiRebalancer

Overview

A support contract for the ZunamiNative token. It sorts the Zunami strategies against mean price across all strategies and rebalances their `lpshares` according to calculated profits.

No issues were found.

5. Conclusion

1 critical, 3 high, 3 medium, 10 low severity issues were found during the audit. 1 critical, 2 medium, 5 low issues were resolved in the update.

The reviewed contracts are highly dependent on the owner’s account. Users using the project have to trust the owner and that the owner's account is properly secured.

This audit includes recommendations on code improvement and the prevention of potential attacks.

We recommend covering the found issues with tests after they are fixed.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY