NAPIER POOL SECURITY AUDIT REPORT

Jul 03, 2024

MixBytes()

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

A group of auditors are involved in the work on the audit. The security engineers check the provided source code independently of each other in accordance with the methodology described below:

1. Project architecture review:

- Project documentation review.
- General code review.
- Reverse research and study of the project architecture on the source code alone.

Stage goals
- Build an independent view of the project's architecture.
- Identifying logical flaws.

2. Checking the code in accordance with the vulnerabilities checklist:

- Manual code check for vulnerabilities listed on the Contractor's internal checklist. The Contractor's checklist is constantly updated based on the analysis of hacks, research, and audit of the clients' codes.
- Code check with the use of static analyzers (i.e Slither, Mythril, etc).

Stage goal

Eliminate typical vulnerabilities (e.g. reentrancy, gas limit, flash loan attacks etc.).

3. Checking the code for compliance with the desired security model:

- Detailed study of the project documentation.
- Examination of contracts tests.
- Examination of comments in code.
- Comparison of the desired model obtained during the study with the reversed view obtained during the blind audit.
- Exploits PoC development with the use of such programs as Brownie and Hardhat.

Stage goal

Detect inconsistencies with the desired model.

4. Consolidation of the auditors' interim reports into one:

- Cross check: each auditor reviews the reports of the others.
- Discussion of the issues found by the auditors.
- Issuance of an interim audit report.

Stage goals

- Double-check all the found issues to make sure they are relevant and the determined threat level is correct.
- Provide the Client with an interim report.

5. Bug fixing & re-audit:

- The Client either fixes the issues or provides comments on the issues found by the auditors. Feedback from the Customer must be received on every issue/bug so that the Contractor can assign them a status (either "fixed" or "acknowledged").
- Upon completion of the bug fixing, the auditors double-check each fix and assign it a specific status, providing a proof link to the fix.
- A re-audited report is issued.

Stage goals

- Verify the fixed code version with all the recommendations and its statuses.
- Provide the Client with a re-audited report.

6. Final code verification and issuance of a public audit report:

- The Customer deploys the re-audited source code on the mainnet.
- The Contractor verifies the deployed code with the re-audited version and checks them for compliance.
- If the versions of the code match, the Contractor issues a public audit report.

Stage goals

- Conduct the final check of the code deployed on the mainnet.
- Provide the Customer with a public audit report.

Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

1.3 Project Overview

Napier is a fixed-rate DeFi protocol that allows to:

- swap large positions on the AMMs for traders;
- get higher APY for liquidity providers.

The audited scope of Napier v1 consists of the implementation of the pool, maintaining liquidity, and interacting with the tranches from the core part and the Curve pools.

1.4 Project Dashboard

Project Summary

Title	Description
Client	Napier Lab
Project name	Napier Pool v1
Timeline	13.12.2023 - 02.07.2024
Number of Auditors	3

Project Log

Date	Commit Hash	Note
12.12.2023	e77d408da81f0bbc96fd71ac710c6cf3a9338ef3	initial commit for the audit
10.04.2024	88d64947dbc677b4b7752f7920b8237d46e46170	commit with the fixes (https://github.com/napierfi/v1-pool/)
16.04.2024	8056387d5bc98c07315f6abcbeb9344eb4eaad67	commit with the fix of bisect condition
30.05.2024	3b1d262a77ee755abd82050c6deb30243f38a08b	commit with the fix of slippage

Project Scope

The audit covered the following files:

File name	Link
src/NapierPool.sol	NapierPool.sol

File name	Link
src/NapierRouter.sol	NapierRouter.sol
src/TrancheRouter.sol	TrancheRouter.sol
src/PoolFactory.sol	PoolFactory.sol
src/libs/PoolAddress.sol	PoolAddress.sol
src/libs/TrancheAddress.sol	TrancheAddress.sol
src/base/Multicallable.sol	Multicallable.sol
src/base/PeripheryImmutableState.sol	PeripheryImmutableState.sol
src/base/PeripheryPayments.sol	PeripheryPayments.sol
src/libs/Constants.sol	Constants.sol
src/libs/Create2PoolLib.sol	Create2PoolLib.sol
src/libs/DecimalConversion.sol	DecimalConversion.sol
src/libs/Errors.sol	Errors.sol
src/libs/PoolMath.sol	PoolMath.sol
src/libs/SignedMath.sol	SignedMath.sol
src/libs/CallbackDataTypes.sol	CallbackDataTypes.sol
src/libs/TrancheMathHelper.sol	TrancheMathHelper.sol#L112

Deployments

File name	Contract deployed on mainnet	Comment

File name	Contract deployed on mainnet	Comment
PoolFactory	0x17354e...e4e4dC61	
Create2PoolLib	0x34a776...e316d9f1	NapierPool bytecode inside
TrancheRouter	0x450B42...34981107	
NapierRouter	0x000000...EeC0cE7d	
NapierPool	0x5a2d32...3c78B821	

1.5 Summary of findings

Severity	# of Findings
Critical	0
High	0
Medium	2
Low	4

ID	Name	Severity	Status
M-1	Insufficient validation of Pool creation parameters	Medium	Fixed
M-2	Incorrect convergence condition	Medium	Fixed
L-1	Incorrect comment	Low	Fixed
L-2	Potential inconsistency while paying with Ether/WETH in the `PeripheryPayments`	Low	Fixed
L-3	Enhanced invariant check in `PoolMath` contract	Low	Fixed
L-4	Excess balances in complex operations of `NapierRouter` not returned to the `payer`	Low	Fixed

1.6 Conclusion

The attack vectors that have been checked during the audit:

1. **Reentrancy**

   - Direct reentrancy attacks targeting `NapierPool` and `NapierRouter` are effectively prevented. This is due to utilizing the `nonReentrant` modifier in all external functions of these contracts.
   - The potential exploitation of read-only reentrancy vulnerabilities derived from using the `CurveTriCrypto` pools is not feasible, as the `calc_token_amount` call is always followed by safeguarded for reentrancy attacks functions. Nevertheless, it's advised not to use on-chain price oracles for these Curve crypto pools while this vulnerability remains.
   - The only external call to an address provided by `msg.sender` is found in the `refundEth` function, which is auxiliary to the primary contract functions.
   - The use of ERC-777 tokens, which could present additional reentrancy vectors, is prohibited by the developers.
   - The concern of cross-contract reentrancy involving interconnected contracts is negated since each `CurveTriCrypto` pool is uniquely linked to a specific `NapierPool`.

2. **Balance Manipulation**

   - The amounts of reserves values within `NapierPool` are handled based on internal fields `totalBaseLpTimesN` and `totalUnderlying18`. These variables are maintained in parallel with the `ERC-20` balance of the contract. All functions affecting balance changes ensure that these balances adjust as anticipated. As a result, the contract logic remains unaffected by any donations occurring between contract function calls.
   - An inflation attack, which would involve artificially increasing the shares ratio, is infeasible. This is due to the initial allocation of `1000` virtual shares during the first `addLiquidity` call, making any such attack attempt financially ineffective for potential attackers.

3. **Front-Running Attacks**

   - The likelihood of successful front-running attacks, particularly those involving price manipulation through flash-loans, is low. This is because all external functions utilize slippage control arguments, except for the specified `Medium 2` scenario. This approach effectively serves as a strong safeguard against such front-run and sandwich attacks.

4. **Data Validation**

   - Crafting of malicious user data is efficaciously prohibited. Throughout the execution flow in `NapierRouter`, there are no calls to externally supplied addresses. The integrity of contract addresses is triple-verified: in `NapierRouter` (ensuring the called `NapierPool` is created by a

recognized factory), in `NapierPool` (confirming the function is called by an authorized callback address), and again in the callback function of `NapierRouter` (verifying the callback is initiated by a specific `NapierPool` address).
- There are comprehensive input data validation checks in both `NapierRouter` and `NapierPool` functions, which effectively prevent the possibility of inconsistent states or blocking core functionalities.

5. **Centralization Risks**

- The contract design minimizes `owner` control, limiting their responsibilities to setting deployment parameters, managing fee-related arguments, and authorizing contracts to interact with `NapierPool`.
- Potential centralization concerns may emerge during the deployment stage. The `owner` is responsible for accurately configuring parameters for `CurveTriCrypto` pools and `NapierPool`, especially concerning Principal Tokens. Given that `NapierPool` adopts the Pendle V2 Protocol model, it's crucial to select Principal Tokens with similar yield rates and risk profiles for a single `CurveTriCrypto` pool. Parameters such as `scalarRoot` and `initialAnchor` need careful calculation based on the expected average yield rates of these tokens.
- The threat of fund theft from `NapierPool` using a new custom caller, maliciously authorized by owner, is impossible due to the strict balance difference validation checks.

6. **Handling Various ERC-20 Token Implementations**

- The contracts employ the `SafeERC20` library for all ERC-20 token transfers, effectively handling most standard token implementations.
- However, specific token types, such as ERC-777, rebaseable tokens, tokens with transfer fees, and tokens with `decimals < 18`, may disrupt the contract functions or lead to inconsistent states.

7. **Price Manipulation**

- The use of `CurveTriCrypto` Liquidity Provider tokens, as opposed to Principal tokens, introduces a risk related to the interdependence of the three tokens. If the yield-generating protocol of one Principal Token is compromised, its liquidity might be exchanged for more stable Principal Tokens. This could result in pools dominated by the compromised token.
- Imbalanced pools are prone to depegged conversion price between LP tokens and Principal Tokens. Consequently, the price ratio of `underlying` tokens to Principal Tokens in such pools might be affected by their market supply, potentially deviating significantly from 1.

8. **Mathematical Model**

- The mathematical model is adapted from Pendle.Finance V2, facilitating efficient liquidity utilization for trades between underlying assets and Principal Tokens, whose prices converge towards 1 as

`maturity` time approaches. However, `NapierPool` uniquely facilliates the trades between `underlying` assets and `CurveTriCrypto` LP tokens across three Principal Tokens.
- Complex mathematical operations like `exp` and `ln` could lead to significant absolute precision errors, especially when dealing with small token amounts. However, these errors do not accumulate, as `NapierPool` maintains the state using total asset amounts and immutable fields like `scalarRoot`, `initialAnchor`, `maturity`, and the dynamically updated `lastLnImpliedRate`, that are not affected by an accumulated error.

9. **Overall Architecture and Code Style**

- The codebase is well-commented, facilitating comprehension despite some minor typos in comments.
- Navigating through the core mathematics algorithms can be challenging, increasing the risk of errors due to the complexity of the underlying model.
- The `NapierRouter` contract, designed for complex trading scenarios, is a bit cumbersome and might be confusing for users. The `swapCallback` function, with numerous conditional branches, can be difficult to follow. An alternative approach could involve using distinct callbacks that are called using a specific `calldata` with signatures, given that callback receivers are pre-authorized by the owner.
- The strict checks in `NapierPool` and `NapierRouter` for validating the user-sent token amounts may restrict some use cases. For instance, adding liquidity to `NapierPool` directly with `CurveTriCrypto` LP tokens is not currently supported, limiting user-friendliness in certain scenarios.

However, there are a few issues that do not significantly impact the overall security of the project (as listed below in section 2).

2.FINDINGS REPORT

2.1 Critical

Not Found

2.2 High

Not Found

2.3 Medium

M-1	Insufficient validation of Pool creation parameters
Severity	Medium
Status	Fixed in d5cc1232

Description

Every Napier Pool is associated with the Curve TriCrypto pool and consists of three principal tokens. There are specific requirements for these tokens to ensure the system functions correctly:

- Those three Principal Tokens MUST have common maturity and underlying with less than or equal to 18 decimals.
- Those three Principal Tokens MUST be deployed by the same TrancheFactory.
- The Underlying MUST be the same as the Underlying of the Principal Token within the Base pool.
- The Base pool MUST be a Curve TriCrypto v2 pool.

Violation of these constraints may cause unexpected behavior.

Related code:

- Pool deployment in PoolFactory: PoolFactory.sol#L50
- constructor of the NapierPool: NapierPool.sol#L107

Recommendation

While these parameters are provided by the system owner and are likely to be validated carefully off-chain, we recommend enhancing the validation of these parameters in the smart contract code to also include on-chain verification.

M-2	Incorrect convergence condition
Severity	Medium
Status	Fixed in 8056387d

Description

In the edge case, the approximation algorithm fails to converge due to unreasonable condition `err_mid < 0` instead of `err_mid <= 0`

Related code: `_bisectUnderlyingNeeded` TrancheMathHelper.sol#L112

Recommendation

We recommend altering the convergence condition.

2.4 Low

L-1	Incorrect comment
Severity	Low
Status	Fixed in 85821e7a

Description

At the line:
PoolMath.sol#L266
the second `netBaseLptToAccount` is likely to actually mean `netUnderlyingToAccount18`.

Recommendation

We recommend fixing the commentary text.

L-2	Potential inconsistency while paying with Ether/WETH in the `PeripheryPayments`
Severity	Low
Status	Fixed in ca00dc7a

Description

In the `_pay` function's code, the smart contract automatically determines the payment method: either to use the ether balance already in the smart contract (likely received through `payable` functions) or to pull `WETH` from the payer using transferFrom. However, the code does not account for the corner case where these payment methods might unintentionally be used simultaneously. For example, the sender pays with `msg.value`, but the `_pay` function ignores it as the value was not large enough and instead pulls `WETH` using `transferFrom`. Consequently, the transferred ether may unintentionally remain in the smart contract and could be withdrawn by a third party using the `refundETH` function.

Currently, we have not found any attack vectors related to this finding, but they may appear during further development of the code.

Related code - the `_pay` function: PeripheryPayments.sol#L82

Recommendation

We recommend adding an assertion that `msg.value` is zero in the code flow that pulls `WETH`.

L-3	Enhanced invariant check in `PoolMath` contract
Severity	Low
Status	Fixed in PR-137

Description

A security concern has been identified in the PoolMath.sol#L219 function within `PoolMath`.

Currently, the function includes a `require` statement that checks if `preTradeExchangeRate != SignedMath.WAD`. However, it is important to note that the value of `preTradeExchangeRate` should not be less than the `WAD` value. To prevent invalid executions that may arise from precision errors in exponent and logarithm calculations, a stricter requirement check can be convenient.

Recommendation

We recommend implementing a `require` check with the `preTradeExchangeRate > WAD` condition. This change will effectively prevent the occurrence of invalid results that could be caused by precision-related errors, thereby enhancing the overall security and reliability of the function.

L-4	Excess balances in complex operations of `NapierRouter` not returned to the `payer`
Severity	Low
Status	Fixed in PR-138, 5d61ec0f

Description

This issue has been found in the `NapierRouter` contract, particularly in functions NapierRouter.sol#L229, NapierRouter.sol#L388, NapierRouter.sol#L451, NapierRouter.sol#L509, NapierRouter.sol#L572, and NapierRouter.sol#L750.

Post-execution of these operations, the `NapierRouter` contract often retains excess balances. Specific concerns for each function are as follows:

- The `swapUnderlyingForPt` function may retain a balance if the `underlying` token is `WETH`.
- In `swapYtForUnderlying`, if the `ptDelta` parameter during the callback exceeds the `ytIn` input argument, only `ytIn` tokens are redeemed, leaving surplus `ptDelta - ytIn` Principal Tokens on the `NapierRouter` contract's balance.
- The `addLiquidity` function sends `pt` tokens to `NapierRouter` before invoking the `NapierPool.addLiquidity` function. Not all of the `CurveTriCrypto` minted `lp` tokens may be utilized, resulting in excess balances of `lp` tokens in `NapierRouter`.
- Similarly, `addLiquidityOnePt` involves partial swapping of `pt` tokens to `underlying` tokens. However, the exact swap proportion is unknown before the transaction's block inclusion, potentially resulting in excess `underlying` or `lp` tokens.
- `addLiquidityOneUnderlying` faces a similar issue as `addLiquidityOnePt`, with the difference being in the transfer and swapping direction of the tokens.
- In `removeLiquidityOnePt`, excess `underlying` tokens may be retained in `NapierRouter` after the liquidity removal from `NapierPool` and token swap operation.

Although retrieval of tokens in these cases is possible via `refundETH` or `sweepToken` using a `Multicallable` interface, direct refunding at the end of the function would be more convenient.

Recommendation

Despite the sweep functions implemented in the router, we recommend automatically returning the excess tokens explicitly at the end of the mentioned functions.

3. ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build opensource solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.

Contacts

https://github.com/mixbytes/audits_public

https://mixbytes.io/

hello@mixbytes.io

https://twitter.com/mixbytes