HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Delorean
Date: May 19, 2023

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Delorean
Approved By	Noah Jelich | Lead Solidity SC Auditor at Hacken OU
Type	DEX; Futures Yield Market
Platform	EVM
Language	Solidity
Methodology	Link
Website	delorean.exchange
Changelog	20.04.2023 - Initial Review 19.05.2023 - Second Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Delorean (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/delorean-exchange/dlx-contracts
Commit	0f21779ac0de4f91256dc1ecb711d3d76e565707
Whitepaper	
Functional Requirements	Link
Technical Requirements	Link
Contracts	File: core/NPVSwap.sol SHA3: 7081f6cee975a176f956d9af987b984562ee40217def8b28971bfca75023deb8 File: core/YieldSlice.sol SHA3: 56b5d81690add771f2c7dbb259f8a904905a9f1e21d45059ad803bdb62510eb5 File: data/Discounter.sol SHA3: 30b452100a0617f0fbc3060971905af19e5e42a111a7d3e2743db7b7159a0c9c File: data/YieldData.sol SHA3: b35eeb750baf6f021450df12b81801979b6f32e3477a3046cf08b712969003c9 File: interfaces/IDiscounter.sol SHA3: 4e62f1d2848bc6f3ffc7476d44784bf7ac816c6118c8cb2d962ecdd76024668c File: interfaces/IGLPRewardTracker.sol SHA3: 845042a7292d5e2f90900ca61dfaf78f90dd96030701d9794e18a11cc5a4bdc6 File: interfaces/ILiquidityPool.sol SHA3: 3b73d05aa137a1fa931118831602ace12736868952204da9d19b91607ce2488b File: interfaces/IYieldSlice.sol SHA3: 9da055dd16ea17e5eb4bcc7b2b605febb0024c262c23f57b19763d2a3a17648a File: interfaces/IYieldSource.sol SHA3: c7525c19126fb7b2f60822e41cc04c7c851095a2ec0d789097efd2de1787e111 File: interfaces/uniswap/IQuoterV2.sol SHA3: ea11eb72f6abe02b260362df550771230069b88c45556a5ea740863ac256a390 File: interfaces/uniswap/ISwapRouter.sol SHA3: f280975d73530056124d74c56b77fa14dbe16e07d1eabcd884d0a9511b60083f

	File: interfaces/uniswap/IUniswapV3Factory.sol SHA3: aedfca34aadf5f9a1e72e1891039f2a119e3fba7a1d7e512cea07e23d573c61e File: interfaces/uniswap/IUniswapV3Pool.sol SHA3: c55b1b0dfdb0f3f14ee1b308782c87d0ecebd7226c436c7f0283c5fb2d596d02 File: sources/StakedGLPYieldSource.sol SHA3: 8acf3d3d1916f22699b462b0a872e55d0e7a6be1f2f0301a1bbbc1dbfaa5c7b8 File: tokens/NPVToken.sol SHA3: 5c2b15fd56173001b5c3ebd85c37e98b5b6034a2de2511bbde5307e8459943af File: liquidity/UniswapV3LiquidityPool.sol SHA3: 89ca835c5d27364b65f60de948feee1f8f8548f09f86a5e742e9b629c633ff19

Second review scope

Repository	https://github.com/delorean-exchange/dlx-contracts
Commit	767fb3182ea8f2aa6a2606be285a776059ce8434
Whitepaper	
Functional Requirements	Link
Technical Requirements	Link
Contracts	File: core/NPVSwap.sol SHA3: 533ed046015a566955b27961a58490be3da5977a1ff23a4b2f664196f30edbbc File: core/YieldSlice.sol SHA3: 81e741b0ce1ab49de63d75b30e2e303cef879f949cfc715e00f50a159e7422c6 File: data/Discounter.sol SHA3: 9908eff3da870a7b3db6914a32a8a8ee88d3d438eda2f548970bd1de6d6af9d6 File: data/YieldData.sol SHA3: e59e50f5ffa975db9f7bc25a4efc40b7a83e16a69884a4b0b2167f1c182bd6a6 File: interfaces/IDiscounter.sol SHA3: d90bb1976244d5dc31ad26a378ac14af36a01cbcaff57651a3389cd4bb2ab726 File: interfaces/IGLPRewardTracker.sol SHA3: 845042a7292d5e2f90900ca61dfaf78f90dd96030701d9794e18a11cc5a4bdc6 File: interfaces/ILiquidityPool.sol SHA3: 11c1934132843d46dd9a3119bab6af6539a4ab3f4971fba86fd62d059a717f07 File: interfaces/IYieldSource.sol SHA3: c7525c19126fb7b2f60822e41cc04c7c851095a2ec0d789097efd2de1787e111 File: interfaces/uniswap/IQuoterV2.sol SHA3: ea11eb72f6abe02b260362df550771230069b88c45556a5ea740863ac256a390 File: interfaces/uniswap/ISwapRouter.sol SHA3: f280975d73530056124d74c56b77fa14dbe16e07d1eabcd884d0a9511b60083f File: interfaces/uniswap/IUniswapV3Factory.sol SHA3: aedfca34aadf5f9a1e72e1891039f2a119e3fba7a1d7e512cea07e23d573c61e File: interfaces/uniswap/IUniswapV3Pool.sol SHA3: c55b1b0dfdb0f3f14ee1b308782c87d0ecebd7226c436c7f0283c5fb2d596d02 File: liquidity/UniswapV3LiquidityPool.sol SHA3: 5fa4b0a3f76e5568ffbcceb73ccf51199da73c6b5d28c5df961131589184ae8f File: sources/StakedGLPYieldSource.sol SHA3: dbc7f74e425d3f1ff4b0581fc7fb87f871146167f48ca1884194cf99c97666db File: tokens/NPVToken.sol SHA3: 7c80078edea0af226be123b74940ab15f69c5584c523f8cf371e9f4f994707e1

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation by external or internal actors.
Medium	Medium vulnerabilities are usually limited to state manipulations but cannot lead to asset loss. Major deviations from best practices are also in this category.
Low	Low vulnerabilities are related to outdated and unused code or minor Gas optimization. These issues won't have a significant impact on code execution but affect code quality

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements:
  - Overall system requirements are provided.
  - Use cases are described and detailed.
- Technical description:
  - Run instructions are provided.
  - Technical specification is provided.
  - NatSpec is satisfactory.

Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.
- Solidity Style Guide is not followed perfectly, but the functions order makes sense.

Test coverage

Code coverage of the project is **100.0%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases covered with tests.
- Interactions by several users are tested thoroughly.

Security score

As a result of the audit, the code contains **1** low severity issue. The security score is 10 out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **10.0**. The system users should acknowledge all the risks summed up in the risks section of the report.

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
20 April 2023	11	2	6	0
19 May 2023	1	0	0	0

Risks

- Some functions within the contracts have nested loops, which could potentially lead to Gas limit issues in specific scenarios.
- Users may face longer lock-up periods for their tokens if the yield rate decreases, reducing their ability to access or trade the underlying tokens.
- The project's contracts interact with external third-party contracts that were not within the scope of this audit. As such, the stability, security, and correct functioning of these external contracts cannot be guaranteed.
- The project's smart contracts allow setting Epoch intervals during deployment. This interval, once set, cannot be changed and can significantly influence the contract's performance on the Arbitrum network.
  A low Epoch interval may lead to a high number of Epochs being created, which can result in computationally intensive loops in contract functions. This could lead to high transaction costs or even risk of the Denial of Service.
  On the other hand, a high Epoch interval may affect the precision of the contract's computations.
  Therefore, it is crucial to carefully select an Epoch interval that ensures efficient functioning of the contract without causing excessive computational load.

System Overview

Delorean is a decentralized finance (DeFi) protocol that focuses on tokens generating real yield. The protocol aims to demonstrate the utility and efficiency of a blockchain-based financial system through its focus on real-yield tokens.

The system enables users to lock yield-generating tokens into debt slices, in exchange for Net Present Value (NPV) tokens. Users can manage their credit positions based on these NPV tokens. The protocol features discounting mechanisms for calculating the present value of future cash flows and yield calculation functions.

The files in the scope:

- **NPVSwap.sol** - the main entry point for the users where they can swap future yield for upfront tokens.
- **YieldSlice.sol** - slice and transfer future yield based on net present value.
- **Discounter.sol** - computes net present value of future yield based on a fixed discount rate.
- **YieldData.sol** - keeps track of historical average yields on a periodic basis. It uses this data to return the overall average yield for a range of time in the `yieldPerTokenPerSlock` method.
- **StakedGLPYieldSource.sol** - wrapper interface for managing yield from sGLP.
- **NPVToken.sol** - NPV tokens are used to track the net present value of future yield.
- **UniswapV3LiquidityPool.sol** - wrapped interface to a Uniswap V3 liquidity pool.
- **IDiscounter.sol** - interface inherited by Discounter.sol, used in YieldSlice.sol.
- **IGLPRewardTracker.sol** - used in StakedGLPYieldSource.sol.
- **ILiquidityPool.sol** - interface for UniswapV3LiquidityPool, inherited by UniswapV3LiquidityPool.sol, used in NPVSwap.sol.
- **IYieldSource.sol** - interface inherited by StakedGLPYieldSource.sol, used in YieldSlice.sol.
- **IUniswapV3Pool.sol** - interface for interacting with UniswapV3Pool, used in UniswapV3LiquidityPool.sol. Inherits 6 files that are out of the scope.
- **IUniswapV3Factory.sol** - interface for interacting with UniswapV3Factory
- **ISwapRouter.sol** - interface for interacting with Uniswap SwapRouter, used in UniswapV3LiquidityPool.sol.
- **IQuoterV2.sol** - interface for interacting with Uniswap QuoterV2, used in UniswapV3LiquidityPool.sol.

Privileged roles

- YieldSlice :
  - Gov :
    - Can set the gov role.
    - Can set the treasury address.
    - Can set the dust limit.
    - Can set the debt fee.
    - Can set the credit fee.
- YieldData :
  - Owner :
    - Can Set the writer address.
  - Writer :
    - Can Record new data.
- Discounter :
  - Owner :
    - Can Set the projected daily yield rate.
    - Can Set the max days of projected future yield to sell.
- StakedGLPYieldSource :
  - Owner :
    - Can Set a new owner.
    - Can Deposit sGLP.
    - Can Withdraw sGLP.
    - Can harvest.
- NPVToken :
  - Owner (YieldSlice.sol):
    - Can mint tokens.
    - Can burn own tokens.

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	Passed
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Failed
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	Passed
Access Control & Authorization	CWE-284	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant
Check-Effect-Interaction	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Passed
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	Not Relevant
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	Passed

Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Not Relevant
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Passed
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155 EIP-712	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant
Shadowing State Variable	SWC-119	State variables should not be shadowed.	Passed
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Passed
Presence of Unused Variables	SWC-131	The code should not contain unused variables if this is not justified by design.	Passed
EIP Standards Violation	EIP	EIP standards should not be violated.	Passed
Assets Integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed
User Balances Manipulation	**Custom**	Contract owners or any other third party should not be able to access funds belonging to users.	Passed
Data Consistency	**Custom**	Smart contract data should be consistent all over the data flow.	Passed

Flashloan Attack	**Custom**	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Passed
Token Supply Manipulation	**Custom**	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed
Gas Limit and Loops	**Custom**	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed
Style Guide Violation	**Custom**	Style guides and best practices should be followed.	Passed
Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed
Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed
Stable Imports	**Custom**	The code should not reference draft contracts, which may be changed in the future.	Passed

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

H01. Highly Permissive Role Access

The owner of the YieldData.sol contract can change the writer from the YieldSlice contract to any desired address. The writer is the only address responsible for recording new data (debt data or credit data). The YieldSlice contract is designed to work with two trackers debtData and creditData, both set during the construction of the YieldSlice contract.

If the owner changes the writer to a malicious or compromised address, unauthorized manipulation of the data could occur, resulting in incorrect yield calculations for both debt and credit sides. This may cause unexpected behavior within the protocol, undermining its overall functionality and reliability.

Path: ./src/data/YieldData.sol : setWriter()

Recommendation: Permit changing the writer for the YieldData contract only once after deployment or implement access control mechanisms such as OpenZeppelin's Ownable and utilize a multi-signature wallet for owner operations to minimize single points of failure. Additionally, consider introducing a timelock for critical owner actions like changing the writer. This would allow the community to review changes and respond accordingly. Providing public documentation on the purpose and usage of this functionality would further enhance transparency and ensure the integrity of the yield tracking process for both debt and credit sides.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

H02. Data Consistency

Using a hardcoded deadline of 1 second (block.timestamp + 1) for a swap on Uniswap V3 leaves very little time for the transaction to be included in a block. It also exposes users to the potential risk of miner timestamp manipulation.

With such a short deadline, there is a higher chance of transactions failing due to network congestion, delays in transaction inclusion, or miner timestamp manipulation. This can cause inconvenience for users, who would need to resend their transactions.

Path: ./src/liquidity/UniswapV3LiquidityPool.sol : swap()

Recommendation: Allow users to pass their own deadline from the frontend, providing them with the flexibility to set a more appropriate deadline based on network conditions and their own risk tolerance. This reduces the risk of failed transactions and potential manipulation while improving the user experience.

Found in: 0f21779

Status: Mitigated (The deadline for transactions was extended from 1 to 10 seconds, reducing the risk of failure. Price fluctuations are managed by user-defined parameters 'amountOutMinimum' and 'sqrtPriceLimitX96', enhancing transaction safety.)

H03. Data Consistency

The transferOwnership function in the YieldSlice contract does not prevent transferring ownership of credit or debt slices to the YieldSlice contract itself or redundant transfers to the current owner.

Additionally, functions debtSlice, mintFromYield, creditSlice and receiveNPV() do not prevent setting the YieldSlice contract as a recipient.

This can lead to potential issues like loss of control, unintended behavior or permanently locked assets.

Path: ./src/core/YieldSlice.sol : transferOwnership(), debtSlice(), mintFromYield(), creditSlice(), receiveNPV()

Recommendation: Create a modifier to check if the recipient is not the address of the YieldSlice contract. Apply this modifier to the functions transferOwnership, debtSlice, mintFromYield, creditSlice, and receiveNPV.

Additionally, add a check for transferOwnership to prevent redundant transfer ownership to the current slice owner.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

H04. Highly Permissive Role Access - Undocumented Behavior

The Gov role in YieldSlice can set the debt fee and the credit fee. These fees are limited to extremely high values (Max debt fee: 50%, Max credit fee: 20%).

There is no documentation about the level of these fees.

Path: ./src/core/YieldSlice.sol

Recommendation: Lower the maximum fees or inform the users about these maximums in the public documentation.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

H05. Denial of Service - Loops Gas Limit

Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.

generatedDebt() and *generatedCredit()* perform a loop that can reach the Gas limit and then revert. Even if these two functions are view functions, they are used by mutative functions; therefore, they can create a Denial of Service.

Path: ./src/core/YieldSlice.sol : generatedDebt(), generatedCredit()

Recommendation: Prevent these loops from reaching the Gas limit.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

H06. Undocumented Behavior

According to the documentation, NPVSwap.sol is supposed to be the entry point contract for the users. However, the users can interact directly with the YieldSlice contract.

Path: ./src/

Recommendation: Align the documentation with the implementation.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

■■ Medium

M01. Non-Finalized Code

The code should not contain forge-std/console.sol imports. The code should be finalized for production.

Paths: ./src/core/NPVSwap.sol

./src/core/YieldSlice.sol

./src/data/YieldData.sol

./src/liquidity/UniswapV3LiquidityPool.sol

./src/sources/StakedGLPYieldSource.sol

Recommendation: Remove unfinalized code, which is only for development purposes.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

M02. Missing Event for Critical Value Update

Critical state changes should emit events for tracking things off-chain.

The functions do not emit events on change of important values.

This may lead to the inability for users to subscribe events and check what is going on with the project.

Paths: ./src/data/Discounter.sol : setDaily(), setMaxDays()

./src/core/YieldSlice.sol : setDebtFee(), setCreditFee(), setGov(), setDustLimit(), setTreasury(), _harvest(), recordData(), debtSlice(), mintFromYield(), transferOwnership()

./src/core/NPVSwap.sol : lockForNPV(), swapNPVForSlice(), lockForYield(), swapForSlice(), mintAndPayWithYield()

./src/sources/StakedGLPYieldSource.sol : setOwner()

./src/data/YieldData.sol : setWriter()

Recommendation: Emit events on critical state changes.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

■ Low

L01. Floating Pragma

The project uses floating pragmas ^0.8.13.

Paths: ./src/core/NPVSwap.sol

./src/core/YieldSlice.sol

./src/data/Discounter.sol

./src/data/YieldData.sol

./src/liquidity/UniswapV3LiquidityPool.sol

./src/sources/StakedGLPYieldSource.sol

./src/tokens/NPVToken.sol

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment.

Found in: 0f21779

Status: Reported

L02. Unused Import

UniswapV3LiquidityPool.sol imports IUniswapV3Factory.sol but does not use it.

Path: ./src/liquidity/UniswapV3LiquidityPool.sol

Recommendation: Remove unused import.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

L03. Interface Mismatch

IYieldSlice.sol is used to represent YieldSlice.sol but is not inherited by it.

Path: ./src/core/YieldSlice.sol

Recommendation: YieldSlice.sol should inherit IYieldSlice.sol.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

L04. Style Guide Violation

The provided projects should follow the official guidelines.

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Modifiers
5. Functions

Functions should be grouped according to their visibility and ordered:

1. constructor
2. receive function (if exists)
3. fallback function (if exists)
4. external
5. public
6. internal
7. private

Within a grouping, place the view and pure functions last.

Constants variables should be in UPPER_CASE_WITH_UNDERSCORES (YieldSlice.unallocId).

Path: ./src/

Recommendation: Follow the official Solidity guidelines.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

L05. Functions that Can Be Declared External

“public” functions that are never called by the contract should be declared “external” to save gas.

Paths: ./src/core/NPWSwap.sol : previewSwapYieldForNPV(), previewSwapYieldForNPVOut(), previewSwapNPVForYield(), previewSwapNPVForYieldOut(), swapNPVForSlice(), previewLockForYield(), previewSwapForSlice(), lockForYield(), swapForSlice(), mintAndPayWithYield()

./src/core/YieldSlice.sol : recordData(), tokens(), remaining()

./src/data/YieldData.sol : yieldPerTokenPerSecond()

Recommendation: Use the external attribute for functions never called from the contract.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

L06. Unused Variables

The variable *nominalGen* is never used.

The variable *deposits* is never used.

Paths: ./src/core/YieldSlice.sol : unlockDebtSlice()

.src/sources/StakedGLPYieldSource.sol : deposits

Recommendation: Remove unused import.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

L07. Missing Zero Address Validation

Address parameters are being used without checking against the possibility of 0x0.

This can lead to unwanted external calls to 0x0.

Paths: ./src/core/YieldSlice.sol : constructor(), setGov(), setTreasury(), debtSlice(), transferOwnership(), creditSlice()

./src/sources/StakedGLPYieldSource.sol : setOwner()

./src/liquidity/UniswapV3LiquidityPool.sol : constructor(), swap()

Recommendation: Implement zero address checks.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

L08. Variable Shadowing

IDiscounter.pv().nominal shadows:

- IDiscounter.nominal()

IDiscounter.nominal().pv shadows:

- IDiscounter.pv()

Path: ./src/interfaces/IDiscounter.sol

Recommendation: Rename related variables/arguments.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

L09. NatSpec Typo

In the NatSpecs of the function *yieldPerTokenPerSecond*(), the fourth parameter is described as “tokens” instead of “yield”.

Path: ./src/data/YieldData.sol : yieldPerTokenPerSecond()

Recommendation: Rename NatSpec parameter.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

L10. NatSpec Contradiction

In the NatSpecs of the function *cumulativeYieldCredit*(), it is specified :

> *Amount of yield generated in the contract's lifetime, exclusive of refunded amounts.*

Instead of subtracting the refunded amounts, the function adds the cumulative paid yield.

Path: ./src/core/YieldSlice.sol : cumulativeYieldCredit()

Recommendation: Provide more explanation about the formula used.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

L11. Repeatable Require Statement

The check if the caller is the owner is repeatable in the StakedGLPYieldSource contract.

Repeating require statements throughout the contract code can lead to unnecessary code duplication. This can make the codebase harder to maintain and more prone to errors.

Path: ./src/sources/StakedGLPYieldSource.sol

Recommendation: Use a modifier instead of repeating require statements. It will make code more maintainable, consistent and readable, while potentially improving Gas efficiency.

Found in: 0f21779

Status: Fixed (Revised commit: 767fb31)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.