MIDGAR

SMART CONTRACT
SECURITY AUDIT OF

1DFX

Price Store Audit

TABLE OF CONTENTS

Project Summary

Security Firm: Midgar

Prepared By: VanGrim, EVDoc

Client Firm: Liquid Lab Company Ltd

Final Report Date: 16 October 2024

Liquid Lab Company Ltd engaged Midgar to review the security of its smart contracts related to the 1DFX platform. From the **9th of September to the 22nd of September**, a team of two (2) auditors reviewed the source code in scope. All findings have been recorded in the following report.

Please refer to the complete audit report below for a detailed understanding of risk severity, source code vulnerability, and potential attack vectors.

Project Name	1DFX
Language	Solidity
Codebase	https://git.liquid-lab.io/1dfx/1dfx-ddex-contracts/
Commit	Initial: 0969420222adde96c1aea399c9c0acc6d48f8c5a Final: 7f15cbd77a3bc99e77610f5ff1ce0a3037ce72e3
Audit Methodology	Static Analysis, Manual Review, Fuzz Testing, Invariant Testing
Review Period	9 September - 22 September 2024
Resolved	16 October 2024

Project Overview

1DFX is a cutting-edge decentralized finance (DeFi) protocol designed to revolutionize the financial ecosystem by leveraging blockchain technology and smart contracts. It addresses key challenges in traditional finance, such as lack of transparency, inefficiency, and centralized control, by providing a robust, transparent, and efficient platform for financial transactions and asset management.

1DFX offers comprehensive deliverable management, including assets, referential assets, index assets, and perpetual futures. The protocol ensures accurate classification, tracking, and management of each deliverable, providing users with a reliable financial ecosystem. It incorporates a rule-based framework for managing deliverables, ensuring all transactions adhere to predefined rules, enhancing security and integrity.

Audit Scope

1DFX-ddex-contracts

ID	File	SHA-1 Checksum
PSF	PriceStoreFacet.sol	**a21aa5049a5ec0bd1198f8985ac77fb20e3ad315**
RDDRF	ReferenceDataDeliverableRulesFacet.sol	**9dd6594058070ad0860a2a1cfccd2bef4e02fb67**
RDDF	ReferenceDataDeliverablesFacet.sol	**bbbce2fb70488220e868f479dc626411662f8357**
RDLP	ReferenceDataLiquidityPoolFacet.sol	**8ff074809299731d42b496aa9aa15a1690488096**
TLP	TranchedLiquidityPoolFacet.sol	**9ff6215c015b498fa090e064a24f8f7080f462d7**
LDT	LibDeliverableTransfer.sol	**f6799f12ae22f074e0fa773a9a806cd7a70abf9d**
LPS	LibPriceStore.sol	**a7ec64623339e98204d55d63364a73bf1e751020**
LRDDR	LibReferenceDataDeliverableRules.sol	**8011d9c9e4982826cde2a3985a1e76b777a4e20e**
LRDD	LibReferenceDataDeliverables.sol	**e93bf0e8d23c0e56d60d135a828e185136dae39a**

1DFX-ddex-contracts

ID	File	SHA-1 Checksum
LRDLP	LibReferenceDataLiquidityPool.sol	**b65df4258529b2d64298827c47029c9a89d9b3b6**
LTLP	LibTranchedLiquidityPool.sol	**54337a6922f9373b3b9962938ce6a996792dcefc**
LTLPS	LibTranchedLiquidityPoolStorage.sol	**3651732ee732e851f38d2ff25109446d0dfc9782**
ONC	OnchainConstants.sol	**a91390006e7f58007d7688b71961170f4c80732a**
GLOBAL	-	**-**

Vulnerability Summary

Vulnerability Level	Total	Acknowledged	Resolved
Critical	0	0	0
High	2	0	2
Medium	5	2	3
Low	3	0	3
Informational	1	0	1

Findings & Resolutions

ID	Title	Severity	Status
LTLPS-1	`deposit()` will revert with panic error if `totalSupply` is greater than `trancheValueUsd`	🟠 **High**	**Resolved**
LTLP-1	Math not rounding in protocol favor when calculating `lptQuantityToMint`	🟠 **High**	**Resolved**
LPS-1	Setting `maxPriceGapBps` to 0 would reject all new differing prices	🟡 **Medium**	**Resolved**
LPS-2	Missing validation when adding or updating `ReferenceRateBoundaries`	🟡 **Medium**	**Resolved**
LPS-3	`submitReferenceRate()` reverts with an underflow error if `maxTimestampVariance` exceeds timestamp	🟡 **Medium**	**Resolved**
PSF-1	Granularity expressed in nanoseconds is not supported in Solidity	🟡 **Medium**	**Acknowledged**
LTLPS-3	`getUSDReferenceRateForDecimals()` with `PRICE_DECIMALS` as an argument returns a truncated price	🟡 **Medium**	**Acknowledged**
LPS-4	The upper and lower bounds of the ranges should be acceptable	🟢 **Low**	**Resolved**
LTLP-2	`estimateWithdrawal()` returns 0 when withdrawal is not allowed	🟢 **Low**	**Resolved**
LPS-5	Recommendations for name changes	🟢 **Low**	**Resolved**
LTLP-3	Avoid using magic numbers in the codebase	🔵 **Informational**	**Resolved**

LTLPS-1		`deposit()` will revert with panic error if `totalSupply` is greater than `trancheValueUsd`	
Asset		LibPriceStore.sol: L101 & L132	
Status		Resolved	
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When both `totalSupply` and `trancheValueUsd` are greater than 0, the LP token price is calculated using the following formula:

```
(uint256(trancheValueUsd) * 1e8) / totalSupply
```

However, if `totalSupply` is greater than `trancheValueUsd`, `_getLptPrice()` will return 0. As a result, `calculateLpTokensToMint()` will revert with a division by zero panic error when calculating `lptQuantityToMint`.

```
calculationResult.lptQuantityToMint = (assetValueInUsdAfterFee * 1e8) / lptPrice;
```

This will cause `deposit()` to be DoS'd until `trancheValueUsd` is greater than or equal to `totalSupply`.

Recommendation

Consider returning `basePrice` when `totalSupply` and `trancheValueUsd` are greater than 0 and `totalSupply` is greater than `trancheValueUsd`.

Resolution

This issue is resolved as of commit 7f15cbd77a3bc99e77610f5ff1ce0a3037ce72e3

LTLP-1		Math not rounding in protocol favor when calculating `lptQuantityToMint`	
Asset		**LibTranchedLiquidityPool.sol: L548**	
Status		**Resolved**	
Rating	**Severity: High**	**Impact: High**	**Likelihood: Medium**

Description

When depositing, the amount of tokens to mint is determined by the following formula:

```
calculationResult.lptQuantityToMint = (assetValueInUsdAfterFee * 1e8) / lptPrice
```

`lptPrice` is set by the `_getLptPrice()` function within `calculateLpTokensToMint()`. If `trancheValueUsd` and `totalSupply` are greater than zero, the price is determined by the following formula:

```
(uint256(trancheValueUsd) * 1e8) / totalSupply
```

However, the price returned by `_getLptPrice()` is rounded down in this case, which is not in favor of the protocol. A lower price would result in more tokens being minted for the depositor.

Recommendation

Use a math library like `FixedPointMathLib` from solady ("@solady/utils/FixedPointMathLib.sol") to round up when it is in protocol favor

Resolution

This issue is resolved as of commit 87fec4691efb7234bacb8f543b682d63f4fdc952

LPS-1	Setting `maxPriceGapBps` to 0 would reject all new differing prices		
Asset	LibPriceStore.sol: L88		
Status	Resolved		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description (POC)

A `maxPriceGapBps` of 0 is allowed when calling `initializePriceStore()` and `setMaxPriceGapBps()`. This means that if the Price Feed Operator submits a new reference rate with a different price, it will be rejected, as `maxPriceGapBps` must be greater than 0 for price changes to be accepted.
As a result, the price won't update until the admin calls `setMaxPriceGapBps()` to modify the `maxPriceGapBps` configuration variable.
This could lead to unexpected losses for traders who didn't have time to manage their positions because the price remained unchanged, and they could face significant losses or even liquidation when the update occurs.

Recommendation

Consider disallowing a `maxPriceGapBps` of 0 in `initializePriceStore()` function.

```
require_ = (maxPriceGapBps < OnchainConstants.ONE_HUNDRED_PERCENT_BPS && maxPriceGapBps > 0);
```

The same check should be performed in `setMaxPriceGapBps()`.

Resolution

This issue is resolved as of commit 8f83afbe9225744b9afee713a3013c6355d89d99

LPS-2	Missing validation when adding or updating `ReferenceRateBoundaries`		
Asset	LibPriceStore.sol: L101 & L132		
Status	Resolved		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description (POC)

When `ReferenceRateBoundaries` are set or updated, the only check existing at the moment ensures that `minimumPrice` and `maximumPrice` are not equal to 0. However, additional checks should be implemented to ensure the Price Feed Operator can update prices. In fact, certain misleading configurations of `minimumPrice` and `maximumPrice` could prevent the Price Feed Operator from updating prices.

- `minimumPrice` equal to `maximumPrice`
- `minimumPrice` greater than `maximumPrice`

In both cases, `submitReferenceRate()` will always revert due to the following check:

```
if (referenceRate.price <= pss.referenceRateBoundaries[referenceRate.baseDeliverableId].minimumPrice ||
                referenceRate.price >=
pss.referenceRateBoundaries[referenceRate.baseDeliverableId].maximumPrice
            ) {
        revert [...]
```

Note that this check would not revert when `minimumPrice` equals `maximumPrice` if the boundaries are included as acceptable prices, as recommended in `LPS-4`

Recommendation

Consider adding a check in `addReferenceRateBoundaries()` and `updateReferenceRateBoundaries()` to ensure that `minimumPrice` is not equal to `maximumPrice` and that `minimumPrice` is less than `maximumPrice`.

Resolution

This issue is resolved as of commit 2d169fdc7b4943e48eab210c1866ff0d393c4bd1

LPS-3		`submitReferenceRate()` reverts with an underflow error if `maxTimestampVariance` exceeds timestamp	
Asset		LibPriceStore.sol L412	
Status		Resolved	
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description (POC)

When`submitReferenceRate()` is called by the price feed operator, two checks are performed on the price timestamp. The first ensures that the new price timestamp is greater than or equal to the previous one. The second ensures that the price timestamp falls within the acceptable variance range defined by `maxPriceTimestampVariance`.
However, the transaction will revert with a panic error due to arithmetic underflow / overflow if `maxPriceTimestampVariance` exceeds the current `block.timestamp` in nanoseconds when the following line of code is executed within `_isPriceTimestampWithinAcceptableVariance()`:

```
newPriceTimestamp <= currentBlockTimestamp - maxPriceTimestampVariance
```

This would prevent the price feed operator from updating the price until the admin adjusts the `maxPriceTimestampVariance`.

Recommendation

Consider checking that `maxPriceTimestampVariance` is less than or equal to `block.timestamp` in nanoseconds within `initializePriceStore()` and `setMaxTimestampVariance()`.

Resolution

This issue is resolved as of commit 21f382a8e2d2ea2689f82c2d6365d1f3918bcfa7

PSF-1	Granularity expressed in nanoseconds is not supported in Solidity		
Asset	PriceStoreFacet.sol		
Status	Acknowledged		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

In `PriceStoreFacet.sol` there are a few functions such as `setMaxTimestampVariance`, `setMaxReferenceRateAge`, `submitReferenceRate`, etc. that takes a parameter that is to be denominated in nanoseconds. However, the lowest time unit allowed in Solidity is 1 second. This means that the setting of nanoseconds inside of the smart contracts does not add granularity to time, but simply scales the number. This adds unnecessary complexity to the validations in `_isPriceTimestampWithinAcceptableVariance()` and `_isPriceStale()` which could lead to incorrect assumption (for example, if the `referenceRate.timestamp` is recorded wrongly.)

Recommendation

Consider removing the nanoseconds conversion completely.

Resolution

1DFX: The granularity used is to maintain consistency with the offchain, so at the moment this will be kept. We did create a new method `_getBlockTimeInNanoseconds` to make things cleaner and easier to understand as part of one of the other mitigations.

Acknowledged and closed.

LTLPS-3		`getUSDReferenceRateForDecimals()` with `PRICE_DECIMALS` as an argument returns a truncated price	
Asset		LibTranchedLiquidityPool.sol	
Status		Acknowledged	
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description (POC)

The price provided by the price feed operator has 10 decimals, however, the price returned by the `getUSDReferenceRateForDecimals()` function has 8 decimals, resulting in a loss of precision.

`getUSDReferenceRateForDecimals()` is called in several functions such as `_getValueOfAssetInTranche()`.

`_getValueOfAssetInTranche()` determines the `trancheValueInUsd` within the `calculateAssetQuantityToWithdraw()` and `calculateLpTokensToMint()` functions. This value is used as an argument when calling the `_getLptPrice()` function.

With a truncated USD reference rate, the `trancheValueUsd` is slightly lower, resulting in a lower price. A lower price means slightly less `assetValueToWithdrawInUsd` in `calculateAssetQuantityToWithdraw()`.

However, the impact is twofold in the case of `calculateAssetQuantityToWithdraw()`, as it results in both lower fees and a higher `lptQuantityToMint` in the following formula:

```
calculationResult.lptQuantityToMint = (assetValueInUsdAfterFee * 1e8) / lptPrice
```

This finding breaks the POOL_03 invariant: The total pool value in USD should be the sum of the USD amounts of the protocol's balances

Recommendation

Use 10 price decimals when calling `getUsdReferenceRateForDecimals()` to calculate `trancheValueInUsd`, or alternatively, call the `getUsdReferenceRate()` function instead.

Resolution

Acknowledged and closed.

LPS-4	The upper and lower bounds of the ranges should be acceptable		
Asset	LibPriceStore.sol: L236-237 & L411-415		
Status	Resolved		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

- Prices equal to either the `minimumPrice` or `maximumPrice` should be permitted. The `ReferenceRateBoundaries` struct defines the boundaries of the acceptable price range, with `minimumPrice` and `maximumPrice` representing these boundaries. However, when `submitReferenceRate()` is called with a `ReferenceRate` struct as an argument, the transaction will revert if the price is equal to either of these boundary values

- Price timestamps equal to either `currentBlockTimestamp - maxPriceTimestampVariance` or `currentBlockTimestamp + maxPriceTimestampVariance` should be permitted Similarly, calling `submitReferenceRate()` will revert if the `newPriceTimestamp` falls on the upper or lower bound of the range defined by `maxPriceTimestampVariance`.

Recommendation

Consider allowing prices that are equal to `minimumPrice` or `maximumPrice` to be included within the acceptable price range.

```
if (referenceRate.price < pss.referenceRateBoundaries[referenceRate.baseDeliverableId].minimumPrice ||
                referenceRate.price >
pss.referenceRateBoundaries[referenceRate.baseDeliverableId].maximumPrice
            ) {
                revert [...]
```

Consider allowing `newPriceTimestamp` to fall on the upper or lower bound of the range defined by `maxPriceTimestampVariance`:

```
if ( newPriceTimestamp < currentBlockTimestamp - maxPriceTimestampVariance || newPriceTimestamp >
currentBlockTimestamp + maxPriceTimestampVariance ) { return true;
```

Resolution

This issue is resolved as of commit cf0bf11ab30fa0c691dcbca0cf4439dbf0272c15

LTLP-2	`estimateWithdrawal()` returns 0 when withdrawal is not allowed		
Asset	LibTranchedLiquidityPool.sol L342		
Status	Resolved		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When `estimateWithdrawal()` is called, if `canWithdraw` is set to false, it returns 0. This is not appropriate because when attempting to withdraw the same `lptQuantityToBurn` as in the view function call, the transaction will revert with a custom error.

Recommendation

Revert with the `TrancheWithdrawalNotAllowed` custom error when `estimateWithdrawal()` is called and `canWithdraw` is set to false.

Resolution

This issue is resolved as of commit 53b899b88704acd40d8ccae637ddbeab0e8031d5

LPS-5	Recommendations for name changes		
Asset	LibPriceStore.sol		
Status	Resolved		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

- `_isPriceTimestampOlder()` checks if `lastPriceTimestamp` is greater than or equal to `nextPriceTimestamp` and returns true if it is. Since updating the price within the same block is meant to reflect real-time changes as accurately as possible, the function should be renamed for better readability. Consider renaming this function to `_isPriceTimestampAcceptable()`

- `_isPriceTimestampWithinAcceptableVariance()` returns true if the `newPriceTimestamp` is outside the acceptable range and false if it is within the acceptable variance. The function name does not accurately reflect its behavior. Consider renaming this function to `_isPriceTimestampOutsideAcceptableVariance()`

- Consider renaming `getLastUSDReferenceRate()` to `getLastRecordedUSDReferenceRate` so that it becomes more descriptive that it can also include stale prices

- The name `blockTimeNs` is misleading as the lowest time unit in solidity is 1 second. But since the unit will be used off-chain it might be better to clarify this with a comment at least.

Resolution

This issue is resolved as of commit 82d88112884ddb159ad595f5c9211327b6398400

LTLP-3	Avoid using magic numbers in the codebase		
Asset	LibPriceStore.sol		
Status	Resolved		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

In `LibTranchedLiquidityPool`, the number 1e10 is used repeatedly throughout the file. In order to avoid confusion, assign a name to the number.

Recommendation

Make sure to assign the number to a fitting name such as `PRICE_DECIMAL_ADJUSTMENT` or `PRICE_SCALE_FACTOR`. in `OnchainConstants.sol`.

Resolution

This issue is resolved as of commit 1ded24acd664bb330e88b29d02963110f14df34b

Appendix

Test Suites: Invariants and Properties

During the security review, Midgar conducted an extensive series of tests on the client's codebase using the Echidna testing tool. The focus was on fuzzing and invariants testing to verify that the properties and invariants within the codebase remained robust under a variety of randomized conditions. This rigorous testing approach was designed to ensure the codebase's resilience and reliability by simulating a wide range of inputs and scenarios.

Over the course of the review, Echidna performed 1,000,000 test runs, systematically exploring potential vulnerabilities and ensuring that the codebase adhered to its specified behaviors.

ID	Description	Run Count	Passed
DEPOSIT_01	Deposit mints LPT to the sender	**1,000,000+**	✅
DEPOSIT_02	Deposit transfers tokens to the protocol	**1,000,000+**	✅
DEPOSIT_03	Deposit increases the tranche value	**1,000,000+**	✅
DEPOSIT_04	Deposit increases the total pool value	**1,000,000+**	✅
WITHDRAW_01	Withdraw deducts LPT from the sender	**1,000,000+**	✅
WITHDRAW_02	Withdraw removes tokens from the protocol	**1,000,000+**	✅
WITHDRAW_03	Withdraw decreases the tranche value	**1,000,000+**	✅
WITHDRAW_04	Withdraw decreases the total pool value	**1,000,000+**	✅
TOKENS_01	Total supply of LPT is equal to the sum of balances of all users	**1,000,000+**	✅

ID	Description	Run Count	Passed
FEES_01	After a deposit, asset sent in USD should be greater than the Lpt minted in USD	**1,000,000+**	✅
FEES_02	After a withdrawal, Lpt burnt in USD should be greater than the amount withdrawn in USD	**1,000,000+**	✅
POOL_01	If the price of assets increase, the total pool value in USD should increase	**1,000,000+**	✅
POOL_02	If the price of assets decrease, the total pool value in USD should decrease	**1,000,000+**	✅
POOL_03	The total pool value in USD should be the sum of the USD amounts of the protocol's balances (LTLPS-3)	**1,000,000+**	❌
DOS	Denial of Service	**1,000,000+**	✅

Vulnerability Classification

The risk matrix below has been used for rating the vulnerabilities in this report. The full details of the interpretation of the below can be seen here.

High Impact	**Medium**	**High**	**Critical**
Medium Impact	**Low**	**Medium**	**High**
Low Impact	**Low**	**Low**	**Medium**
	Low Likelihood	**Medium Likelihood**	**High Likelihood**

Methodology

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by aspiring auditors.

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Midgar to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Midgar’s position is that each company and individual are responsible for their own due diligence and continuous security. Midgar’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Midgar are subject to dependencies and are under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access and depend upon multiple layers of third parties.

Notice that smart contracts deployed on the blockchain are not resistant to internal/external exploits. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Midgar does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

About Midgar

Midgar is a team of security reviewers passionate about delivering comprehensive web3 security reviews and audits. In the intricate landscape of web3, maintaining robust security is paramount to ensure platform reliability and user trust. Our meticulous approach identifies and mitigates vulnerabilities, safeguarding your digital assets and operations. With an ever-evolving digital space, continuous security oversight becomes not just a recommendation but a necessity. By choosing Midgar, clients align themselves with a commitment to enduring excellence and proactive protection in the web3 domain.

To book a security review, message https://t.me/vangrim1.

MIDGAR