PUBLIC

Code Assessment

of the Limit Order Settlement Smart Contracts

January 10, 2023

Produced for

by

Contents

1 Executive Summary

Dear 1inch team,

Thank you for trusting us to help 1inch with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Limit Order Settlement according to Scope to support you in forming an opinion on their security risks.

In Limit Order Settlement resolvers settle orders of users. Major advantages this system offers include MEV protection and gasless swaps for the creator of the order. Resolvers should be whitelisted, in order to join this whitelist sufficient stake of 1inch tokens must be allocated to the resolver. The staking and delegation make use of the new proposed ERC20Pods extension.

The most critical subjects covered in our audit are functional correctness, security of the assets and the accounting of the balances.

The general subjects covered are design, efficiency and documentation. While the Settlement system may protect from MEV done by the block producers, orders may be observed/rearranged on another level. The staking is only used as a barrier of entry and does not ensure that a resolver follows the protocol rules as stated in the documentation.

Detailed documentation / specification and documentation explaining the interactions between the components, especially with the limit order protocol was largely missing during the review. This review was done based on our understanding of the system as in the System Overview of this report for which we did not receive a confirmation of 1inch.

In summary, we find that the codebase in its current state provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1
Low-Severity Findings	2
• Code Corrected	1
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

This review covers the 1inch Limit Order Settlement Smart Contract, an extension to the 1inch Limit Order Protocol. The assessment was performed on the source code files inside `contracts` folder of the Limit Order Settlement repository based on the documentation files. Following files from the repository contracts folder were part of the assessment scope:

```
helpers/VotingPowerCalculator.sol
interfaces/IFeeBank.sol
interfaces/IFeeBankCharger.sol
interfaces/IResolver.sol
interfaces/ISettlement.sol
interfaces/IVotable.sol
interfaces/IWhitelistRegistry.sol
libraries/DynamicSuffix.sol
libraries/OrderSaltParser.sol
BasicDelegationPodWithVotingPower.sol
FeeBank.sol
FeeBankCharger.sol
RewardableDelegationPodWithVotingPower.sol
Settlement.sol
St1inch.sol
WhitelistRegistry.sol
```

in (Version 2) the following files were added:

```
helpers/ResolverMetadata.sol
helpers/St1inchPreview.sol
helpers/WhitelistHelper.sol
```

in (Version 3) the following files were added:

```
libraries/Address.sol
```

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	22 November 2022	eba5d2c1fff015b3b00ded8f67abf05490dc4c48	Initial Version
2	2 December 2022	a0e73b9cc7545b0cdd85ee6dccf95b1ceeb3c586	Updated Version
3	19 December 2022	515e4b777535b484c785ff5e5acc2875777ef22b	Version 3

For the solidity smart contracts, the compiler version `0.8.17` was chosen.

2.1.1 Excluded from scope

The bases BasicDelegationPod and RewardableDelegationPod as well as ERC20Pods have been reviewed separately and are not part of this report.

The limit order protocol itself (which the settlement contract interacts with) is not part of the audit scope.

Any contracts not mentioned above, mock and testing contracts that might rely on the scoped contracts are not part of the scope. Imported libraries are assumed to behave according to their specification and are not part of the assessment scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

1inch offers a new system for limit orders settlement. A major issue with limit orders is front running, Limit Order Settlement addresses this problem by allowing trusted *resolvers* to batch orders with the help of the `Settlement` contract. The makers also do not have to pay gas for their orders, as the resolvers will take care of it, however, they still must pay a fee. The resolvers act as takers for every order in the batch, aggregating all the makers assets in one contract, and then can optimize the matching between the orders. The incentives for resolvers are gas cost and fees included in each order's rate.

A whitelist is employed to highlight the supposedly trusted resolvers. In order to be included in this whitelist, resolvers must stake `1inch` tokens, from 1 day to 4 years, to get a decreasing-over-time voting power that will allow them to stay in the top list.

FeeBank

This contract allows resolvers to deposit and withdraw 1inch tokens that will be charged as fees upon order settlement. Another contract, `FeeBankCharger` will be responsible for the accounting of the charged fees. The `FeeBank` contract holds the funds and the internal accounting tracks the deposited amounts for each account and updates the available credit on the `FeeBankCharger` contract upon deposit/withdrawal. The available functions are:

- `deposit`: allows the caller to deposit `1inch` tokens in the contract. The internal accounting of deposits and the available credit on the `FeeBankCharger` contract will be increased by the deposited amount. The tokens must be approved beforehands.
- `depositFor`: similar to `deposit`, but the specified `account` will be credited in place of the caller. The tokens must be approved beforehands.
- `depositWithPermit`: similar to `deposit`, but does not require prior approval.
- `depositForWithPermit`: similar to `depositFor`, but does not require prior approval.
- `withdraw`: allows the caller to withdraw its available credit. The function will revert if the `amount` is greater than the available credit in the `FeeBankCharger` contract.
- `withdrawTo`: similar to `withdraw`, but the funds are sent to the specified `account`.
- `gatherFees`: the owner of the contract can call this function collect the fees from a list of accounts.

FeeBankCharger

The `FeeBankCharger` contract is responsible for tracking and updating the available credit for each resolver. It will deploy it own `FeeBank` at deployment. The contract does not hold funds, but the `FeeBank` will rely on the internal accounting of the available credit to distribute the fees. The contract is intended to be inherited from in order to provide all its functionalities. The state changing functions are:

- `increaseAvailableCredit`: only the `FeeBank` can call this function. The function increases the available credit of the target account by the given amount.
- `decreaseAvailableCredit`: only the `FeeBank` can call this function. The function decreases the available credit of the target account by the given amount.
- `_chargeFee`: this function is internal. It allows to charge a fee for a given account by decreasing its available credit. The function will revert if the account does not have enough credit.

Settlement

The `Settlement` contract inherits from `FeeBankCharger` and allows resolvers to optimize the settlement of batched orders by recursively filling each order up to the `IInteractionNotificationReceiver.fillOrderInteraction()` callback in `OrderMixin.fillOrderTo()`, find matching orders and optimize the swaps, and then finish the settlement of the batch. In this setting, the resolver aggregates all the making assets and amounts before optimizing the swaps and transfering the taker assets and amounts to the makers in the batch. The resolvers will have to pay a fee for using the system, to this end, the `Settlement` contract is `FeeBankCharger` and the resolvers will need to have enough credit in the `FeeBank` to execute the orders. For each recursive call, a suffix is appended to the end of the interactive data to track the total fee, that will be charged at the end. The contract exposes two functions:

- `settleOrders`: this is the entry point function for resolvers. To resolve orders in batches, they must build the calldata such that it will recursively trigger the settlement of each order in the batch
- `fillOrderInteraction`: this function is a callback from the `OrderMixin` and can only be called by the limit order protocol. This is where the recursive calls happen, depending on the interactive data, the control flow either settles the next order or gives the control to a `Resolver` contract at the end of the batch processing.

The data for `settleOrders` must have a specific layout. It must be the calldata that would be sent if `OrderMixin.fillOrderTo` was called, but without the function selector. In the case of batched orders, orders must be included in the interactive data part of the previous order.

First, for each `Order`, *makers* can use the salt field to encode some information:

- bits 255-224: starting time for a Dutch auction
- bits 223-192: duration time for a Dutch auction
- bits 191-176: initial rate bump
- bits 175-144: order fee
- bits 0-143: order salt

With this encoded information, makers create a Dutch auction for their order, where the order's rate decreases (from `100%+initial rate bump` to `100%`) the incentive for the resolvers grows as time passes, but from a game theoretical point of view, the Dutch auction makes them compete against each other to execute the orders with a rate that is still acceptable for the makers.

Then, the *makers* must include a list of whitelisted resolver addresses they trust for settling their orders. This list must be at the end of `Order.interaction` bytes array and have the following format: `address0|address1|...|addressN|length`. The `length` is the number of whitelisted addresses in the list and must fit on one byte.

Resolvers must also encode the interactive data in a way the control flow can be redirected either to continue the recursive order settlement after the callback to `fillOrderInteraction()`, or to give the control flow to the `Resolver` contract. To indicate whether there solver wants to finalize the interaction, a special byte `0x01` must be added after the `interactionTarget` address, any other value will continue the interaction (recursion).

Layout of `data`, example for 2 orders. order1 and order2 refer to the `Order` struct of the limit order protocol. Padding is not displayed in this example:

```
order1.offset (32bytes)
sig1.offset (32bytes)
resolverInteractionsOrder1.offset (32bytes)
makingAmount for order1 (32bytes)
takingAmount for order1 (32bytes)
skipPermitAndThresholdAmount for order1 (32bytes)
target for order1 (32bytes)
order1.salt (32bytes, with special encoding described above)
order1.makerAsset (32bytes)
. . . (other order1 members omitted for brevity, 32bytes each)
order1.interactionOffset (32bytes)
order1.interactionLength (32bytes)
order1.standardInteraction (order1.interactionLength bytes)
order1.whitelistedAddresses (20bytes each)
order1.whitelistedAddressesLength (1byte)
length of sig1 (32bytes)
sig1 (length of sig1 bytes)
length of resolverInteractionsOrder1 (32bytes)
resolverInteractionsOrder1 (length of resolverInteractionsOrder1 bytes)
    interactionTargetOrder1 (20bytes, should be the Settlement contract)
    0x00 (1byte, we don't want to finalize the interaction yet)
    order2.offset (32bytes)
    sig2.offset (32bytes)
    resolverInteractionsOrder2.offset (32bytes)
    makingAmount for order2 (32bytes)
    takingAmount for order2 (32bytes)
    skipPermitAndThresholdAmount for order2 (32bytes)
    target for order2 (32bytes)
```

```
    order2.salt (32bytes with special encoding described above)
    order2.makerAsset (32bytes)
    . . . (other order2 members omitted for brevity, 32bytes each)
    order2.interactionOffset (32bytes)
    order2.interactionLength (32bytes)
    order2.standardInteraction (order2.interactionLength bytes)
    order2.whitelistedAddresses (20bytes each)
    order2.whitelistedAddressesLength (1byte)
    length of sig2 (32bytes)
    sig2 (length of sig2 bytes)
    length of resolverInteractionsOrder2 (32bytes)
    resolverInteractionsOrder2 (length of resolverInteractionsOrder2 bytes)
        interactionTargetOrder2 (20bytes, should be the Settlement contract)
        0x01 (1byte, we want to finalize the interaction)
        resolver contract address (20bytes)
        some data for the resolver
```

St1inch

Holders of the 1inch token can stake their tokens in this contract. By staking, users lock their stake for a certain amount of time (between 1 day and 4 years), withdrawal is only possible thereafter. Voting power per staked 1inch token depends (non-linear, exponentially) on the time the token is locked for. The calculation of votingPowerAt/balanceAt is an approximation and not exact. The values tracked by `balances`, i.e. `_balanceAt(X, T) / _VOTING_POWER_DIVIDER` represent the amount that should have been put in at `VotingPowerCalculator` origin time, to have a voting power of `X * _VOTING_POWER_DIVIDER%` at the target deadline `T`. So every balance has the same origin point in time and thus `balance1 > balance2 <==> VP1(T) > VP2(T)` for a given timestamp `T`.

- `deposit/depositWithPermit/depositFor/depositForWithPermit`: Allows to deposit tokens for a certain lock duration. Calling this function with an amount of zero may be used to extend the duration of the lock and hence increases voting power for this user. If these functions are used to deposit additional stake, the lock up time is increased for the whole stake
- `increaseLockDuration`: Wrapper for deposit with an amount of 0, increases the lock duration and hence voting power
- `increaseAmount`: Allows to deposit more
- `withdraw/withdrawTo`: Allows to withdraw the stake, withdrawal is only possible once the lock period is over or emergency exit has been activated

There is an emergency exit function which can be enabled/disabled by the owner.

- `setEmergencyExit`: Allows the owner to toggle the boolean flag if emergency exit is active or not

Several view functions allow retrieval of account balance or voting power. All transfer / allowance functionality has been disabled.

This contract implements the ERC20Pods extension. This extension informs (calls `pod.updateBalances()` in case of any balance change of accounts registered to a pod). Pods are used to track delegations (e.g. for the whitelist or governance). These delegations are independent as they are handled by different pods. Notably an address can delegate its stake for one resolver regarding the whitelist while at the same time delegate its stake to another account for governance purposes.

In addition to the delegation pods 1inch may add a pod for farming rewards / to incentivize the staking.

BasicDelegationPod and RewardableDelegationPod

These pods allow to track delegations. The WhitelistRegistry will use one to track the voting power of the accounts. Furthrmore the Governance may use one to track delegation for Governance purpose.

Users wishing to delegate must register the pod in the St1Inch contract and call `delegate()` on the Pod.

Basic Delegation Pod:

This pod allows a user to delegate his balance to another account using function `delegate`. Balances delegated to an account are tracked by minting or burning ERC20 tokens accordingly for this delegatee. These tokens are used for accounting only, transfers and approvals have been disabled.

Rewardable Delegation Pod:

Extension of BasicDelegationPod. Allows delegatees to distribute rewards to users having delegated to them. When delegating the base ERC20Pods token users receive delegation share tokens (which also supports the ERC20Pods standard). On this token they register for the delegatees Rewards pod (called farm contract) which is in charge to distribute the actual rewards.

Delegatees must first register within the RewardableDelegationPod. Only after a delegatee has register user can delegate to this delegatee.

Upon balance updates, first the accounting for the delegated amount is updated (see BasicDelegationPod), in addition the user delegating gets delegation shares minted.

These contracts extend the base contracts of the Delegating repository, namely they each have an additional `votingPowerOf` function.

WhitelistRegistry

Using one of the pods described above the voting power of an account is accessed. This pod is set as the token for the WhitelistReigstry, the WhitelistRegistry will then query this token for the balance or voting power.

- `register`: Allows any account to register for the whitelist. To register successfully one must have at least a voting power of `resolverThreshold`. If there is no space left in the whitelist the poorest account gets removed (if the balance of the account registering exceeds this accounts balance).
- `clean`: Permissionless function, removes any account with a voting power below the threshold from the whitelist.
- `promote`: Allows any msg.sender to register an addresss per chainId (uint256). This allows whitelisted entities to register the actual executor address on every supported chain.

Owner functionality:

- `setResolverThreshold`: Allows to set the minimum voting power a resolver must have. Using the permissionless function `clean`, addresses in the whitelist below this threshold can be removed.
- `setWhitelistLimit`: Allows to set the maximum size of the whitelist. If the new size exceeds the current amount of whitelisted address, the poorest accounts are removed front he whitelist.
- `rescueFunds`: Allows the Owner to rescue any token or Ether balance at this contract.

Users should only craft and sign orders including whitelisted resolvers. Whether a resolver is part of the whitelist is not checked within the settlement smart contract. While building the order users should query the whitelist and choose their trusted resolvers. One may has to make sure the whitelist is up to date e.g. using `clean()` or by assessing otherwise whether all resolver still have sufficient delegated stake.

This contract is intended to be deployed only on mainnet, each resolver can set their corresponding address on other supported chains in this contract, so resolvers need to stake only on mainnet and this contract can also serve as a whitelist for other chains, which can be queried with `getPromotees(chainId)`.

Changes in Version 2

The following functional changes were introduced in version 2:

St1inch:

- Stakers can now withdraw early using `earlyWithdrawTo`. This incurs a penalty fee, the function features some protections (maxLoss / minReturn) to ensure the user accepts amount paid out. Furthermore this loss is limited by the maxLossRatio set by the owner. The fee collected (which is in 1inch tokens) is transferred to the FeeReceipient address set by the admin.
- `rescueFunds` allows the owner to rescue locked Ether or tokens. In case of the 1inch token the owner can only rescue the surplus, not balance belong to the stake.

Settlement:

- The suffix of `orderInteractions` now additionally contains the deadline (starting **Version 3** this is called public availability timestamp). If the deadline is exceeded `checkResolver` returns true, meaning any address can act as resolver.

The following new helper contracts exist:

- ResolverMetadata: Allows resolvers registered in the delegation contract passed in the constructor to set their resolver Url in this contract.
- St1inchPreview: Implements `previewBalance(address account, uint256 amount, uint256 duration)`. Returns the balances of St1inch token a user receives additionally when staking this amount for duration. If the account has an unlock date already later than duration, this is taken into account.
- WhitelistHelper: Implements a view function `getMinAmountForWhitelisted()`, returns the minimum amount needed to join the whitelist.

Changes in Version 3

The following functional changes were introduced in version 3:

St1inch:

- Functions `increaseLockDuration` and `increaseAmount` have been dropped.
- The minimum lock period has been increased from 1 day to 30 days.
- A default farm can be set for the `St1inch` ERC20Pods token.

Settlement:

- A new taking fee has been introduced. The fee is set by the maker and is a percentage of the taking amount that the resolver can take. The maker must encode the fee receiver and ratio after the whitelisted addresses and the deadline.
- On top of specifying which resolvers are allowed to fill their order, makers need to add a timestamp for each allowed resolver to specify until when each of them is allowed to settle the order.

2.2.1 Trust Model and Roles

Resolvers: fully untrusted. From a game theoretical perspective, thanks to the Dutch auction, users are protected from getting the worst price as long as there is at least one honest whitelisted resolver.

Owner Role in contracts (e.g. St1inch, WhitelistRegistry) fully trusted

1inch's frontend: all the orders that are meant to be used by the `Settlement` system are assumed to be created through 1inch's frontend, that is trusted to make the users sign on private orders, i.e., `allowedSender==Settlement`, and set the whitelisted resolvers.

1inch's backend: trusted for checking the validity of the orders it receives, e.g., order is private, resolvers are whitelisted, ... Moreover, everybody can query and see the orders from the backend.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Staking Does Not Prevent Misbehavior Risk Accepted

Low-Severity Findings	1

- Missing Events Acknowledged

5.1 Staking Does Not Prevent Misbehavior

Design Medium Version 1 Risk Accepted

Resolvers have to join a whitelist which is governed by the staking of 1inch tokens.

The documentation states:

```
The stake determines a resolver’s ability to get orders and ensures that a resolver
follow the protocol rules (like in proof of stake model).
```

On the smart contract level the implementation of the staking does not allow to seize stake of bad actors. Their stake is not at risk and can simply be withdrawn at the end of the lock period hence this staking does not ensure that a resolver follows the protocol rules.

Risk accepted:

1inch states:

```
They'll need only follow what is required to be able to settle the order batch.
Staking is only used as a threshold entry requirement.
```

5.2 Missing Events

Design Low Version 1 Acknowledged

Events are used to be informed of or to keep track of transactions changing the state of a contract. Generally, any important state change should emit an event.

The functions used for deposits and withdrawals in `FeeBank` do not emit an event, hence it's hard for an observer to track deposits and withdrawals

---

Acknowledged:

1inch acknowledged the issue and decided to leave the code as it is.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- St1inch Can Be Locked Indefinitely Code Corrected

Low-Severity Findings	1

- Resolver Can Set Arbitrary Callback Code Corrected

6.1 `St1inch` Can Be Locked Indefinitely

Security Medium Version 1 Code Corrected

It is possible for an attacker to lock the staked amount of `1inch` token of any staker by using one of the `St1inch.depositFor` functions for the target address. By depositing a small amount of tokens and specifying the duration, one can force a target staker to see its stake locked for more time, preventing the staker to withdraw. The only way to break that attack would be to activate the emergency exit to allow the target staker to withdraw.

Code corrected:

The functions `St1inch.depositFor` and `St1inch.depositForWithPermit` have been updated so the duration cannot be specified and is hardcoded to be `0`. This will only increase the deposited amount and not the timelock duration.

6.2 Resolver Can Set Arbitrary Callback

Security Low Version 1 Code Corrected

Resolvers can set the callback address (`interactionTarget` address) called in `Settlement._settleOrder()` and execute arbitrary code which may severely interfere with the process.

Code corrected:

The Settlement contract now ensures that the address is the settlement contract itself:

```
let target := shr(96, calldataload(add(data.offset, interactionOffset)))
if iszero(eq(target, address())) {
    mstore(0, errorSelector)
```

```
    revert(0, 4)
}
```

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Competing Resolvers May Result in Failing Transactions

Informational **Version 1**

Since resolvers are competing against each other, it may happen that more than one resolver submits the same order in their respective batch, in the same block. In such cases, only the first batch including the order will not revert and all the other resolvers will suffer from pure loss of gas.

7.2 Gas Optimization

Informational **Version 1**

Some operations can be in an `unchecked` block to save gas, examples are:

- update of `i` and addition in `FeeBank.gatherFees()`
- addition in `FeeBank._depositFor()`
- addition in `FeeBankCharger.increaseAvailableCredit()`
- `for` loop in `WhitelistRegistry.register()`
- `WhitelistRegistry._shrinkPoorest()`

Intermediary memory variable can save storage reads. Example is:

- `St1inch._deposit()` does two SLOAD for `deposits[account]`, storing the updated deposit amount in memory will save gas.

Code partially corrected:

The function `St1inch._deposit()` has been updated to do only one SLOAD for the depositor.

Other gas optimizations have been addressed in future commits.

7.3 Preview Functions Accept Invalid Durations

Informational **Version 1**

The preview functions (`previewBalance`, `previewPowerOf`,`previewPowerOfAtTime`) may accept a duration parameter that may exceed the maximum locking period and make the transaction revert if applied in the `St1inch` contract.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Allowed Sender of Orders

Note Version 1

Makers wishing to benefit from the protections of the limit settlement protocol must ensure the order they sign has the settlement contract set as allowed sender.

Technically the settlement contract allows resolvers to batch any orders which gives them greater freedom to aggregate transactions. While execution of orders without the allowed sender restricted works, such orders can also be executed through the limit order protocol directly and hence lack the protection limit settlement order offers.

It's vital to understand that this field has to be set correctly or that the protections offered by limit order settlement don't apply. Although this might be obvious there should be documentation emphasizing this. Even the tests within the limit-settlement-order repository use public orders (since allowed sender is not set and hence anyone, not just the settlement contract, can call `limitOrderProtocol.fillOrder()` for this order).

This is an easy source of errors, hence it`s important to be explicit and not assume users/integrators will understand and do this correctly.

8.2 User Responsibility for Setting Trusted Resolvers

Note Version 1

Nothing enforces the resolvers listed in `Order.interaction` to be actually part of the `WhitelistRegistry`. It is the user's responsibility to ensure that the resolvers addresses they sign over are trusted.

1inch stated:

```
That’s also the responsibility of the frontend to provide correct whitelists to the user.
And also responsibility of the backend to filter out maliciously created orders without
the proper whitelist.
```