SETTLEMENT PROTOCOL SMART CONTRACT AUDIT REPORT FOR 1INCH

DEC.22

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading web3 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Coinstats, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

KASPER ZWIJSEN

Lead Smart Contract Auditor | Hexens

Audit Starting Date	Audit Completion Date
15.12.2022	29.12.2022

---

hexens × 1inch NETWORK

METHODOLOGY

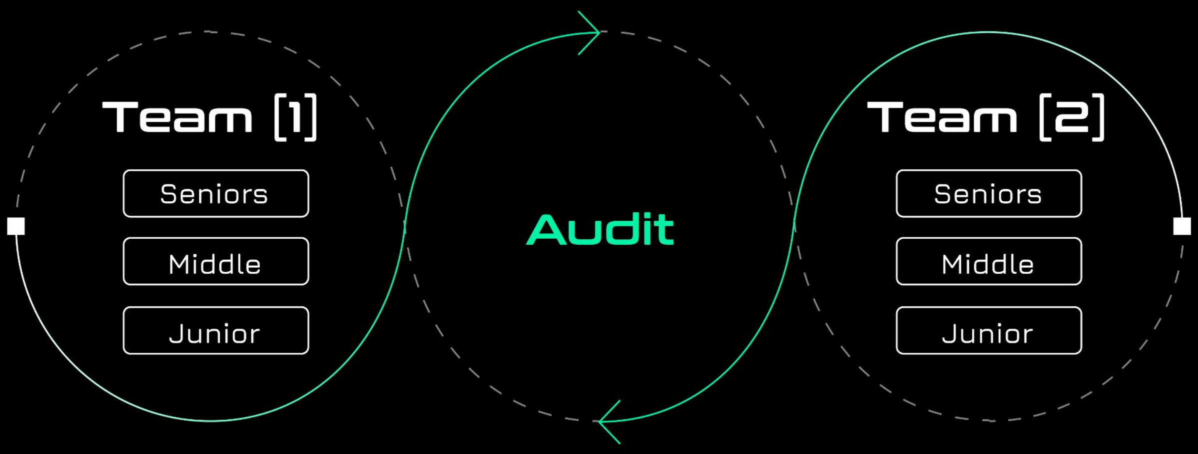
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor*
Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of smart contracts. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

The newly developed projects by 1inch include ERC20Pods, limit order settlements and a new delegation system. The latter two are both build upon the ERC20Pods contracts.

Our security assessment was a full review of the projects and its smart contracts. We have thoroughly reviewed each contract individually, as well as the system as a whole.

During our audit, we have identified 1 critical vulnerability in the ERC20Pods contracts and as a result both St1inch and the delegation system were affected. The vulnerability would allow an attacker to gain an arbitrary amount of voting power.

We have also identified various minor vulnerabilities and optimisations.

Finally, all of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:
https://github.com/1inch/limit-order-settlement/commit/0e8189419c5fbbc67416a1d9e7721cb06b51a38b
https://github.com/1inch/erc20-pods/commit/2b4545d435e159f08ddf80fd14e8f4efb665bcdb
https://github.com/1inch/delegating/commit/9a243d64422c41b0631617466deeb85dcd92e57c

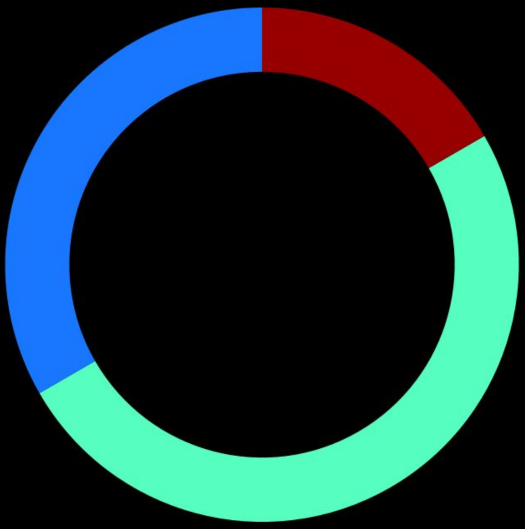
The issues described in this report were fixed. Corresponding commits are mentioned in the description.

SUMMARY

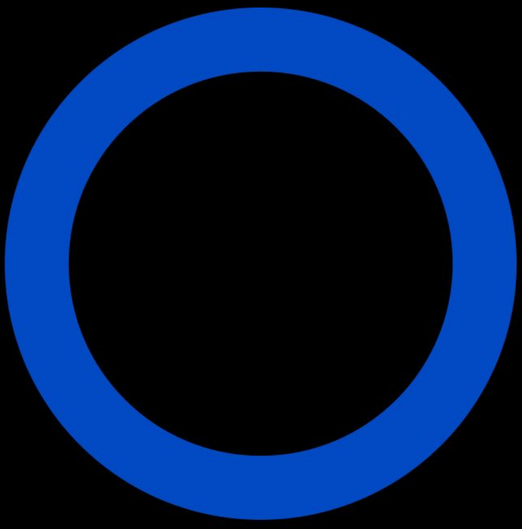
SEVERITY	NUMBER OF FINDINGS
CRITICAL	1
HIGH	0
MEDIUM	0
LOW	3
INFORMATIONAL	2

TOTAL: 6

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

1. INFINITE VOTING POWER

SEVERITY: Critical

PATH: ERC20Pods.sol:_updateBalances:115-132

REMEDIATION: increase the gas limit in St1inch.sol (e.g. from 500K to 530K) or to decrease the gas limit in DelegatedShare.sol (e.g. from 150K to 140K). This would leave enough gas to execute the external call

STATUS: fixed

DESCRIPTION:

The **1inch** staking contract **St1inch.sol** implements **ERC20Pods.sol** and therefore supports attaching pods for delegating voting power, such as the pod **PowerPod.sol**. A user can add a pod for themselves by calling **ERC20Pods.sol:addPod**. The pod will receive an external call upon every token transfer to correspondingly update their balances, as implemented in **ERC20Pods.sol:_afterTokenTransfer** and **Pod.sol:updateBalances**.

This external call in **ERC20Pods.sol:_updateBalances** is limited in gas and implemented using an external call in Yul (assembly). Because of the assembly call, the return code (i.e. whether the external was successful or not) is ignored. This ultimately means that any state changing effects that happened before or will happen after the external call will still go through.

Another important aspect is that **PowerPod.sol** requires the user to call **PowerPod.sol:register**, which creates a **DelegatedShare.sol** contract. This contract also implements **ERC20Pods.sol** and further allows adding delegation pods. The **ERC20Pods.sol:addPod** function allows any address to be added as a pod, including attacker-controlled contracts, but only to oneself.

The **St1inch.sol** contract has a limit of 500K gas for the external call to the **PowerPod.sol** and the **DelegatedShare.sol** has a limit of 150K gas for each registered pod. Furthermore, the **DelegatedShare.sol** has a maximum of 3 pods per user. This means that a user can register 3 pods that burn all 150K gas and leave 50K gas for all of the operations in **PowerPod.sol**. This is insufficient and it is therefore possible to force a revert of the external call from **St1inch.sol** to **PowerPod.sol**.

This vulnerability can be exploited to generate infinite voting power on **PowerPod.sol** and **DelegatedShare.sol** contract. Basically, when a **St1inch.sol** balance update happens, the attached **PowerPod.sol** and its **DelegatedShare.sol** should also update the user's balance correspondingly. However, the external call can be forcefully reverted and the result code is ignored. As a result, the balance in **St1inch.sol** is updated, but the balance in **PowerPod.sol** is not.

The following scenario shows a more practical example:

1. A user has 1 1INCH
2. The user calls **PowerPod.sol:register**, which creates a **DelegatedShare.sol** for the user
3. The user calls **PowerPod.sol:delegate** and delegates to themselves
4. The user calls **St1inch.sol:addPod** with the address of the RDP pod
5. The user calls **St1inch.sol:deposit** to deposit their 1 1INCH and receives 1 ST1INCH (1:1 for simplicity's sake)
6. The deposit triggered **St1inch.sol:_afterTokenTransfer**, which called **updateBalances** on the RDP. At this moment, the user has a balance of 1 ST1INCH, 1 RDP and 1 DS
7. Now the user calls **DelegatedShare.sol:addPod** with a random address, e.g. **address(1)**. This will cause the **DelegatedShare.sol:_afterTokenTransfer** and consequently the **PowerPod.sol:updatesBalances** to always revert
8. Afterwards the user calls **St1inch.sol:withdraw** for 1 1INCH (instant withdrawals for simplicity's sake)
9. The withdrawal triggered **St1inch.sol:_afterTokenTransfer**, which calls the always reverting **updateBalances** on the RDP, but the transaction still succeeds. At this moment, the user has a balance of 0 ST1INCH, 1 RDP and 1 DS
10. The user calls **DelegatedShare.sol:removePod** with the same address as step 7
11. The user calls **St1inch.sol:deposit** again for 1 1INCH
12. Now the user has a balance of 1 ST1INCH, 2 RDP and 2 DS
13. Repeat step 7-12 for infinite voting power

ERC20Pods.sol:

```
function _updateBalances(address pod, address from, address to, uint256 amount)
private {
  bytes4 selector = IPod.updateBalances.selector;
  bytes4 exception = InsufficientGas.selector;
  uint256 gasLimit = podCallGasLimit;
  assembly {  // solhint-disable-line no-inline-assembly
      let ptr := mload(0x40)
      mstore(ptr, selector)
      mstore(add(ptr, 0x04), from)
      mstore(add(ptr, 0x24), to)
      mstore(add(ptr, 0x44), amount)

      if lt(div(mul(gas(), 63), 64), gasLimit) {
          mstore(0, exception)
          revert(0, 4)
      }
      pop(call(gasLimit, pod, 0, ptr, 0x64, 0, 0))
  }
}
```

2. ARITHMETIC GAS OPTIMISATION

SEVERITY: Low

PATH: St1nch.sol:_withdraw:L202

REMEDIATION: in St1nch.sol:_withdraw on line 202 depositor.unlockTime should be set directly to block.timestamp as this will save gas and does not change the lock/unlock time logic

STATUS: fixed

DESCRIPTION:

In the function **St1nch.sol:_withdraw** on line 202 the unlock time of the deposit is set to the minimum of the original unlock time and the current timestamp. However, this arithmetic operation is redundant and the unlock time of the deposit should be set directly to the current timestamp.

If it is an early withdrawal or a withdrawal at exactly the unlock time, then the unlock time will be set to the current timestamp and in the next deposit in the function **St1nch.sol:_deposit** on line 129 the maximum operation will return the current timestamp.

If it is a late withdrawal, then the unlock time will be set to the original unlock time, but in the next deposit the maximum on line 129 will always be the current timestamp.

In all cases the calculated unlock time on line 129 will return the current timestamp.

St1nch.sol:

```
function _withdraw(Depositor memory depositor, uint256 amount, uint256 balance)
private {
  totalDeposits -= amount;
  depositor.amount = 0;
  // keep unlockTime in storage for next tx optimization
  depositor.unlockTime = uint40(Math.min(depositor.unlockTime, block.timestamp));
  depositors[msg.sender] = depositor; // SSTORE
  _burn(msg.sender, balance);
}
```

3. MISSING ZERO ADDRESS CHECKS

SEVERITY: Low

PATH: St1inch.sol:setFeeReceiver (L70-73), FeeBank.sol:_depositFor (L104-108)

REMEDIATION: add a check against the zero address in St1inch.sol:setFeeReceiver for the parameter feeReceiver_ and in FeeBank.sol:_depositFor for the parameter account

STATUS: fixed, commits: 1, 2

DESCRIPTION:

1. In **St1inch.sol:setFeeReceiver** the parameter **feeReceiver_** is not checked against the zero address. If this address is set to **address(0)**, then the early withdrawal function would always revert.
2. In **FeeBank.sol:_depositFor** the parameter **account** is not checked against the zero address. This zero address could be propagated from **depositFor** and it would increase the balance of **address(0)**.

St1inch.sol:

```
In St1inch.sol:setFeeReceiver the parameter feeReceiver_ is not checked against the zero address. If this address is set to address(0), then the early withdrawal function would always revert.

In FeeBank.sol:_depositFor the parameter account is not checked against the zero address. This zero address could be propagated from depositFor and it would increase the balance of address(0).
```

FeeBank.sol:

```
function _depositFor(address account, uint256 amount) internal returns (uint256 totalAvailableCredit) {
  _token.safeTransferFrom(msg.sender, address(this), amount);
  _accountDeposits[account] += amount;
  totalAvailableCredit = _charger.increaseAvailableCredit(account, amount);
}
```

4. EARLY WITHDRAWAL IS IMPOSSIBLE IF THE FEE RECEIVER IS NOT SET

SEVERITY: Low

PATH: St1inch.sol:earlyWithdrawTo:L155-170

REMEDIATION: either add a check to redirect the fees to the current contract in case the fee receiver is not set, since a rescueFunds function is implemented and those tokens will be able to be withdrawn afterwards. Or to add the feeReceiver address as a parameter in the constructor and set the variable so that it is always set

STATUS: fixed

DESCRIPTION:

The function **St1inch.sol:earlyWithdrawTo** is designed to give users the opportunity to withdraw earlier than the staking period end time. However, in the case where the **feeReceiver** address has not yet been set, then there will be no way to make an early withdrawal as the **feeReceiver** address will be zero address, and the **transfer** function call will revert.

This will be the case when the contract has been created as the **feeReceiver** is not a parameter in the constructor.

St1inch.sol:

```
function earlyWithdrawTo(address to, uint256 minReturn, uint256 maxLoss) public {
  Depositor memory depositor = depositors[msg.sender]; // SLOAD
  if (emergencyExit || block.timestamp >= depositor.unlockTime) revert StakeUnlocked();
  uint256 allowedExitTime = depositor.lockTime + (depositor.unlockTime - depositor.lockTime) *
minLockPeriodRatio / _ONE;
  if (block.timestamp < allowedExitTime) revert MinLockPeriodRatioNotReached();

  uint256 amount = depositor.amount;
  if (amount > 0) {
     uint256 balance = balanceOf(msg.sender);
     (uint256 loss, uint256 ret) = _earlyWithdrawLoss(amount, balance);
     if (ret < minReturn) revert MinReturnIsNotMet();
     if (loss > maxLoss) revert MaxLossIsNotMet();
     if (loss > amount * maxLossRatio / _ONE) revert LossIsTooBig();

     _withdraw(depositor, balance); console.log("ret:", ret); console.log("loss:",loss);
console.log("feeReceiver: ",feeReceiver);
     oneInch.safeTransfer(to, ret);
     oneInch.safeTransfer(feeReceiver, loss);
  }
}
```

5. INCOMPLETE INTERFACES

SEVERITY: Informational

PATH: IDelegatedShare.sol (L7), IERC20Pods.sol (L7), IPod (L5)

REMEDIATION:

1. import IERC20Pods.sol and make IDelegatedShare inherit from IERC20Pods
2. add podsLimit and podCallGasLimit as external view functions to IERC20Pods.sol
3. add token as an external view function to IPod.sol
4. add feeBank as an external view function to IFeeBankCharger.sol
5. add availableCredit, depositFor, depositWithPermit, withdraw, withdrawTo and gatherFees as external functions to IFeeBank.sol

STATUS: fixed, commits: 1, 2, 3

DESCRIPTION:

1. The interface **IDelegatedShare.sol** is an interface for **DelegatedShare.sol**, which implements ERC20Pods.sol. However, the interface does not inherit from **IERC20Pods.sol** and is therefore missing all the public functions of **ERC20Pods.sol**.
2. The interface **IERC20Pods.sol** is missing functions for the public variables **podsLimit** and **podCallGasLimit**.
3. The interface **IPod.sol** is missing a function for the public variable **token**.

4. The interface **IFeeBankCharger.sol** is missing a function for the public variable **feeBank**.

5. The interface **IFeeBank.sol** is missing functions for the public functions **availableCredit**, **depositFor**, **depositWithPermit**, **withdraw**, **withdrawTo** and **gatherFees**.

IDelegatedShare.sol:

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface IDelegatedShare is IERC20 {
  function addDefaultFarmIfNeeded(address account, address farm) external; // onlyOwner
  function mint(address account, uint256 amount) external; // onlyOwner
  function burn(address account, uint256 amount) external; // onlyOwner
}
```

IERC20Pods.sol:

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

interface IERC20Pods is IERC20 {
  event PodAdded(address account, address pod);
  event PodRemoved(address account, address pod);

  function hasPod(address account, address pod) external view returns(bool);
  function podsCount(address account) external view returns(uint256);
  function podAt(address account, uint256 index) external view returns(address);
  function pods(address account) external view returns(address[] memory);
  function podBalanceOf(address pod, address account) external view returns(uint256);

  function addPod(address pod) external;
  function removePod(address pod) external;
  function removeAllPods() external;
}
```

IPod.sol:

```
pragma solidity ^0.8.0;

import "./interfaces/IPod.sol";

abstract contract Pod is IPod {
   error AccessDenied();

   address public immutable token;

   modifier onlyToken {
      if (msg.sender != token) revert AccessDenied();
      _;
   }

   constructor(address token_) {
      token = token_;
   }
}
```

IFeeBankCharger.sol:

```
pragma solidity 0.8.17;

interface IFeeBankCharger {
   function availableCredit(address account) external view returns (uint256);
   function increaseAvailableCredit(address account, uint256 amount) external returns (uint256);
   function decreaseAvailableCredit(address account, uint256 amount) external returns (uint256);
}
```

IFeeBank.sol:

```
pragma solidity 0.8.17;

interface IFeeBank {
   function deposit(uint256 amount) external returns (uint256);
}
```

6. REDUNDANT IMPORTS

SEVERITY: Informational

PATH: RewardableDelegationPod.sol (L8), RewardableDelegationPodWithVotingPower.sol (L6, L7)

REMEDIATION: remove the redundant import in RewardableDelegationPod.sol (L8), RewardableDelegationPodWithVotingPower.sol (L6, L7)

STATUS: fixed, commits: 1, 2

DESCRIPTION:

1. In **RewardableDelegationPod.sol** on line 8, **IDelegatedShare.sol** is imported. However, this contract has already been imported on line 6.
2. In **RewardableDelegationPodWithVotingPower.sol** on line 6 and line 7, **VotingPowerCalculator.sol** and **IVotable.sol** are imported. However, these contracts will already be imported on line 8.

RewardableDelegationPod.sol:

```
import "./interfaces/IDelegatedShare.sol";
```

RewardableDelegationPodWithVotingPower.sol:

```
import "./helpers/VotingPowerCalculator.sol";
import "./interfaces/IVotable.sol";
```

hexens