hexens × 1inch NETWORK

SETTLEMENT PROTOCOL SMART CONTRACT AUDIT REPORT FOR 1INCH

NOV.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

VAHE KARAPETYAN

Co-founder / CTO | Hexens

Audit Starting Date
17.10.2022

Audit Completion Date
04.11.2022

---

hexens × 1inch NETWORK

METHODOLOGY

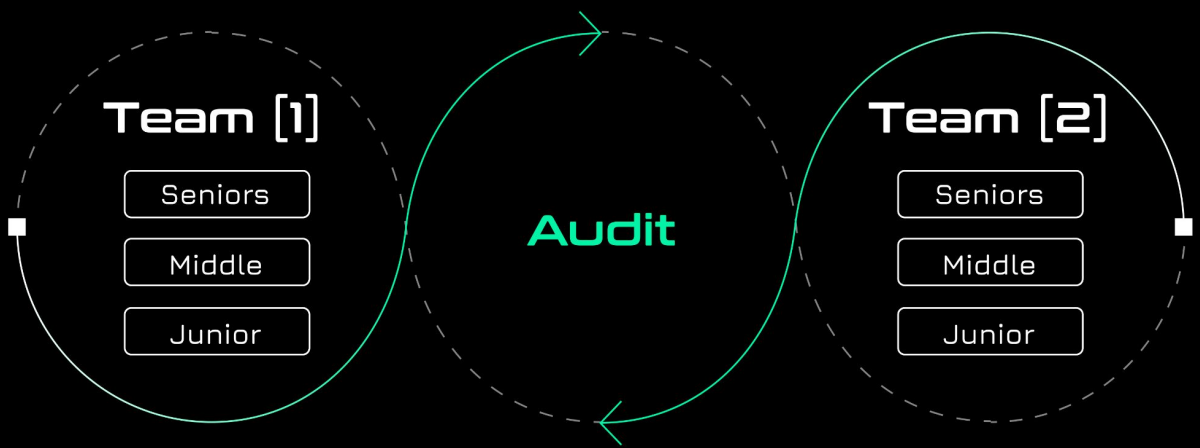
COMMON AUDIT PROCESS

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 2 critical vulnerabilities in the ERC20Pods contracts, hence subtle reentrancy issues which allow to hijack the exectionion flow in between state changes. The vulnerabilities would allow an attacker to do double delegation and thus amplify their voting power.

Also two high severity vulnerabilities were identified that allowed blockage of the protocol's specific components.

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/erc20-pods/commit/50b192a7e63d4eb4eaa8c8af431feb6113e620fd
https://github.com/1inch/delegating/commit/c8f42022216458ba0425fae5c710c79d9ea8bb71
https://github.com/1inch/limit-order-settlement/commit/117fac4b93ce21846ef78a8554c4fcd2524481df

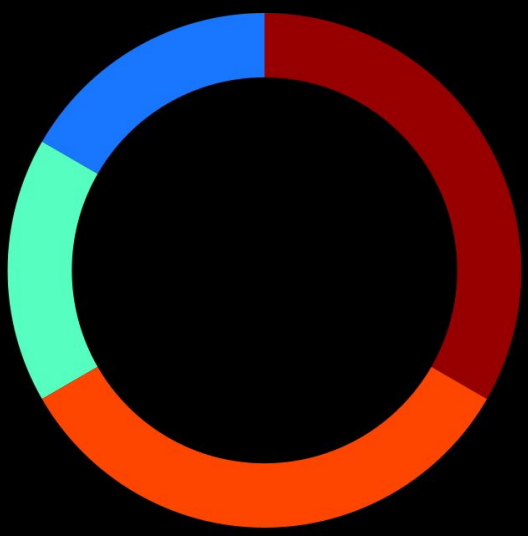
The issues described in this report were fixed. Corresponding commits are mentioned in the description.

SUMMARY

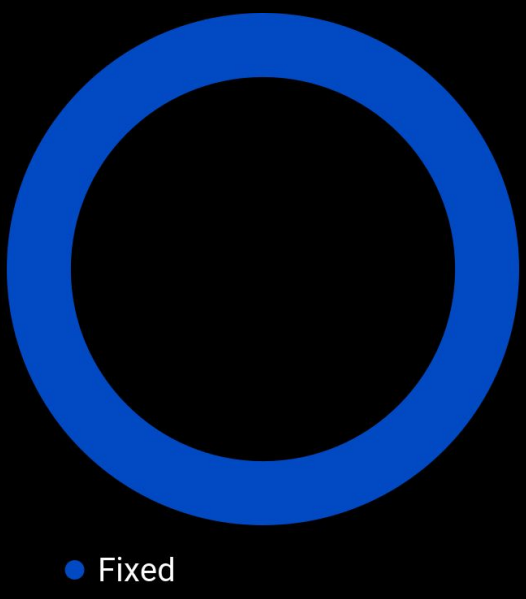
SEVERITY	NUMBER OF FINDINGS
CRITICAL	2
HIGH	2
MEDIUM	0
LOW	1
INFORMATIONAL	1

TOTAL: 6

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

1. CROSS-FUNCTION REENTRANCY LEADING TO DOUBLE DELEGATION

SEVERITY: Critical

PATH: ERC20Pods.sol:116-151

REMEDIATION: consider reconstructing the logic in the _afterTokenTransfer hook instead, also using reentrancy locks can be effective for cross-function reentrancy miitgation

STATUS: fixed

DESCRIPTION:

In the function **_beforeTokenTransfer** that is being overloaded to process the delegations transfers for various pods, user pods are being read from **_pods** mapping and then the code checks whether the "**from**" and "**to**" addresses share or not the pods, the logic can be described as:

1. When some pod is shared by both parties – call **pod.updateBalances(from,to,amount)** effectively transferring delegation from → to
2. Pod is used only by "from" – burn the delegation for "from" by calling **pod.updateBalances(from, 0, amount)**
3. Pod is delegated only by "**to**" – mint the delegation for "**to**" by calling **pod.updateBalances(0,to,amount)**

The assumption here is that user pod arrays cannot be changed during the call, although in the case when "**from**" is by itself a fake pod contract and "**to**" has only the valid pod; "from" can abuse the **updateBalances** call in step 2 to reenter to call **ERC20Pods.addPod()** function with parameter "**pod**" set as the valid pod, thus changing the _pods mapping and minting itself valid delegation tokens, after the calls ends the _beforeTokenTransfer hook will continue processing the old array of pods, thus effecting double mint of delegation to "**to**" in step 3.

By the end of the attack both "**from**" and "**to**" will have valid delegation topic tokens, although "**to**" must have been the only beneficiary of the delegation.

ERC20Pods.sol:

```
function _beforeTokenTransfer(address from, address to, uint256 amount) internal override virtual {
    super._beforeTokenTransfer(from, to, amount);

    if (amount > 0 && from != to) {
      address[] memory a = _pods[from].items.get();
      address[] memory b = _pods[to].items.get();

      for (uint256 i = 0; i < a.length; i++) {
        address pod = a[i];

        uint256 j;
        for (j = 0; j < b.length; j++) {
          if (pod == b[j]) {
            // Both parties are participating of the same Pod
            _updateBalances(pod, from, to, amount);
            b[j] = address(0);
            break;
          }
        }

        if (j == b.length) {
          // Sender is participating in a Pod, but receiver is not
          _updateBalances(pod, from, address(0), amount);
        }
      }

      for (uint256 j = 0; j < b.length; j++) {
        address pod = b[j];
        if (pod != address(0)) {
          // Receiver is participating in a Pod, but sender is not
          _updateBalances(pod, address(0), to, amount);
        }
      }
    }
  }
```

2. REENTRANCY LEADING TO DOUBLE DELEGATION

SEVERITY: Critical

PATH: ERC20Pods.sol: 85-96

REMEDIATION: update the balance in the loop or use balanceOf in _updateBalances function

STATUS: fixed

DESCRIPTION:

In the function **_removeAllPods** contract iterates over the delegated pods and burns the delegation balances via calling **updateBalances** on pods, the bug arises since the balance is saved once on line 87, and an illicit actor can use both a valid pod and a rogue pod. Thus the first call to the rogue pod can be used to make a highjack execution flow and change the balance between the pod removals.

Attack scenario:

1. The illicit actor sends out almost all his tokens to a vault contract (pre-deployed for the attack), such that he's left with only 1 token (1/10**decimals of the token) on its balance
2. Adds both valid and rogue pods via calling **addPod**
3. Calls **removeAllPods**
4. **balance** variable is initialised with value 1, and the rogue pod will be called first to update the balances
5. Rogue pod hijacks the flow and transfers back all of the tokens back to the illicit actor
6. The next iteration will call the valid pod with balance = 1 instead of the new balance, so the actor's pod balance will be changed only by 1, and the pod will be removed
7. The illicit actor again calls **addPod** with the valid pod, which mints pod tokens.

ERC20Pods.sol:

```
function _removeAllPods(address account) internal virtual {
    address[] memory items = _pods[account].items.get();
    uint256 balance = balanceOf(account);
    unchecked {
      for (uint256 i = items.length; i > 0; i--) {
        if (balance > 0) {
          _updateBalances(items[i - 1], account, address(0), balance);
        }
        _pods[account].remove(items[i - 1]);
      }
    }
  }
```

3. FRONTRUNNING TO BLOCK USER WITHDRAWAL

SEVERITY: High

PATH: St1inch.sol: 118-134, 145-165

REMEDIATION: the duration of the lock should be updatable only by the owner, e.g. depositFor and depositForWithPermit should call _deposit with 0 duration

STATUS: fixed

DESCRIPTION:

The functions **withdraw** and **withdrawTo** can be frontrunned by any illicit actor by calling **depositFor** or **depositForWithPermit** and passing the victims account and 0 as amount and any lock period as duration (e.g. MAX_LOCK_PERIOD). Since the function **_deposit** ignores token transfer logic in case of 0 amount and just adds the lock duration. Any user withdrawal requests can be blocked using such attack vector.

St1inch.sol:

```
function depositFor(
    address account,
    uint256 amount,
    uint256 duration
) external {
    _deposit(account, amount, duration);
}

function depositForWithPermit(
    address account,
    uint256 amount,
    uint256 duration,
    bytes calldata permit
) public {
    oneInch.safePermit(permit);
    _deposit(account, amount, duration);
}
```

St1inch.sol:

```
  function _deposit(
      address account,
      uint256 amount,
      uint256 duration
  ) private {
      if (_deposits[account] > 0 && amount > 0 && duration > 0) revert
ChangeAmountAndUnlockTimeForExistingAccount();

      if (amount > 0) {
          oneInch.transferFrom(msg.sender, address(this), amount);
          _deposits[account] += amount;
          totalDeposits += amount;
      }

      uint256 lockedTill = Math.max(_unlockTime[account], block.timestamp) +
duration;
      uint256 lockedPeriod = lockedTill - block.timestamp;
      if (lockedPeriod < MIN_LOCK_PERIOD) revert LockTimeLessMinLock();
      if (lockedPeriod > MAX_LOCK_PERIOD) revert LockTimeMoreMaxLock();
      _unlockTime[account] = lockedTill;

      _mint(account, _balanceAt(_deposits[account], lockedTill) /
_VOTING_POWER_DIVIDER - balanceOf(account));
  }
```

4. BLOCKING THE MINIMAL REGISTRANT IN THE WHITELIST

SEVERITY: High

PATH: WhitelistRegistry.sol:49-70

REMEDIATION: implement a check for AddressSet.add() return value

STATUS: fixed

DESCRIPTION:

The function **register** is used to register a new resolver if the voting power meets the threshold requirement, in case the whitelist is full (**_whitelist.length() == maxWhitelisted**) the function needs to find minimal registrant (with minimal voting power) and remove him from the _whitelist set (**AddressSet**).

In the case where an already registered resolver calls the function, the contract will try to add the resolver once more and will delete the minimal registrant. The bug arises because **AddressSet**.add does not revert but returns "**false**" instead and there is no check implemented for this case.

As a result, any registrant except for the one with minimal voting power is able to "block"/remove the minimal registrant even in the case when the **_whitelist.length() == maxWhitelisted**

WhitelistRegistry.sol:

```
function register() external {
    if (token.votingPowerOf(msg.sender) < resolverThreshold) revert
BalanceLessThanThreshold();
    uint256 whitelistLength = _whitelist.length();
    if (whitelistLength < maxWhitelisted) {
      _whitelist.add(msg.sender);
      return;
    }
    address minResolver = msg.sender;
    uint256 minBalance = token.balanceOf(msg.sender);
    for (uint256 i = 0; i < whitelistLength; ++i) {
      address curWhitelisted = _whitelist.at(i);
      uint256 balance = token.balanceOf(curWhitelisted);
      if (balance < minBalance) {
        minResolver = curWhitelisted;
        minBalance = balance;
      }
    }
    if (minResolver == msg.sender) revert NotEnoughBalance();
    _whitelist.remove(minResolver);
    _whitelist.add(msg.sender);
    emit Registered(msg.sender);
  }
```

5. ARBITRARY EXTERNAL CALL

SEVERITY: Low

PATH: Settlement.sol: 96-102

REMEDIATION: it is recommended to avoid arbitrary calls and use selector/signature-based calldata construction, as well as if possible implement filtering for the call targets

STATUS: fixed

DESCRIPTION:

In the function **fillOrderInteraction** that is being called as a callback within **OrderMixin** (limit-order-protocol) the parameter **interactiveData** is being parsed as arrays of targets and calldata with the purpose of making arbitrary external calls. There were no impactful vectors found for the reviewed implementation, although it is recommended to avoid arbitrary calls for the sake of security in case the code will be refactored in the future or the protocol evolves in its composability.

```
        uint256 length = targets.length;
        if (length != calldatas.length) revert IncorrectCalldataParams();
        for (uint256 i = 0; i < length; i++) {
            // solhint-disable-next-line avoid-low-level-calls
            (bool success, ) = targets[i].call(calldatas[i]);
            if (!success) revert FailedExternalCall();
        }
```

6. REDUNDANT CHECK

SEVERITY: Informational

PATH: WhitelistRegistry.sol:114

REMEDIATION: implement a correct shrinking algorithm

STATUS: fixed

DESCRIPTION:

The function **_shrinkPoorest** there is a condition **if (i < addresses.length)** at **WhitelistRegistry.sol:_shrinkPoorest()** which is redundant due to the same check at line 114.

WhitelistRegistry.sol:

```
function _shrinkPoorest(AddressSet.Data storage set, IVotable vtoken, uint256 size)
private {
    uint256 richestIndex = 0;
    address[] memory addresses = set.items.get();
    uint256[] memory balances = new uint256[](addresses.length);
    for (uint256 i = 0; i < addresses.length; i++) {
        balances[i] = vtoken.balanceOf(addresses[i]);
        if (balances[i] > balances[richestIndex]) {
            richestIndex = i;
        }
    }

    for (uint256 i = size; i < addresses.length; i++) {
        if (balances[i] <= balances[richestIndex]) {
            // Swap i-th and richest-th elements
            (addresses[i], addresses[richestIndex]) = (addresses[richestIndex], addresses[i]);
            (balances[i], balances[richestIndex]) = (balances[richestIndex], balances[i]);

            // Find new richest in first size elements
            if (i < addresses.length) {
                richestIndex = 0;
                for (uint256 j = 1; j < size; j++) {
                    if (balances[j] > balances[richestIndex]) {
                        richestIndex = j;
                    }
                }
            }
        }
    }
```