Smart Contract Security Audit Report

1inch Fusion v2

Decurity, 2023

Contents

1. General Information

This report contains information about the results of the security audit of the 1inch (hereafter referred to as “Customer”) Fusion protocol v2 smart contracts, conducted by Decurity in the period from 03/27/2023 to 03/31/2023.

1.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

1.2. Scope of Work

The audit scope included the contracts in the following repository tree: https://github.com/1inch/limit-order-settlement/tree/feature/next-gen. Initial review was done for the commit 580ba89e10ea54da983cd5a4de7326b511da75d0. The re-testing was done for the commit f8bc44868a6e6dfaf070510b4ffb928c9b8ee473.

The following contracts have been tested:

- Settlement.sol,
- libraries/DynamicSuffix.sol,
- libraries/FusionDetails.sol.

1.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract). The centralization risks have not been considered upon the request of the Customer.

The main possible threat actors are:

- User,
- Protocol owner,
- Limit Orders Protocol,
- Resolver.

The table below contains sample attacks that malicious attackers might carry out.

Table. Theoretically possible attacks

Attack	Actor
Contract code or data hijacking *Deploying a malicious contract or submitting malicious data*	Contract owner Token owner
Financial fraud *A malicious manipulation of the business logic and balances, such as a re-entrancy attack or a flash loan attack*	Anyone
Attacks on implementation *Exploiting the weaknesses in the compiler or the runtime of the smart contracts*	Anyone

1.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

1.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

2. Summary

As a result of this work, we haven't discovered any critical exploitable security issues..

Our suggestions included fixing the low-risk issues and some best practices (see 3.1).

2.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of Apr 3, 2023.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Unnecessary address check	FeeBank.sol	Info	Fixed
Calldata location can be used instead of memory	FeeBank.sol	Info	Fixed
Non optimal reading storage variables	WhitelistRegistry.sol	Info	Fixed
Non optimal decrement/increment	WhitelistRegistry.sol	Info	Acknowledged
Integer overflow (not exploitable)	FeeBank.sol	Info	Acknowledged
Unnecessary ADD operator	Settlement.sol	Info	Fixed

3. General Recommendations

This section contains general recommendations on how to fix discovered weaknesses and vulnerabilities and how to improve overall security level.

Section 3.1 contains a list of general mitigations against the discovered weaknesses, technical recommendations for each finding can be found in section 4.

Section 3.2 describes a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level.

3.1. Current Findings Remediation

Follow the recommendations in section 4.

3.2. Security Process Improvement

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

4. Findings

4.1. Unnecessary address check

Risk Level: Info

Status: Fixed in the commit f8bc44868a6e6dfaf070510b4ffb928c9b8ee473.

Contracts:

- FeeBank.sol

Description:

There is an unnecessary charger address check on the 23rd line:

```
contracts/FeeBank.sol:
23:  if (address(charger_) == address(0)) revert ZeroAddress();
```

FeeBank is created in the constructor of FeeBankCharger.sol and the first argument can't be equal to zero.

```
contracts/FeeBankCharger.sol:
  21      constructor(IERC20 token) {
  22:         feeBank = new FeeBank(this, token, msg.sender);
  23      }
```

Remediation: Consider removing the unnecessary check.

4.2. Calldata location can be used instead of memory

Risk Level: Info

Status: Fixed in the commit f465b95f31d3f8603965c372160a51c3bbc8d2f6.

Contracts:

- FeeBank.sol

Description:

`accounts` variable is not modified during the execution of the `gatherFees()` function so its location can be changed to `calldata`.

```
contracts/FeeBank.sol:
   99:       function gatherFees(address[] memory accounts) external
onlyOwner returns (uint256 totalAccountFees) {
```

Remediation: Consider changing the location of the `accounts` array to calldata instead of memory to save gas.

4.3. Non optimal reading storage variables

Risk Level: Info

Status: Fixed in the commit f8bc44868a6e6dfaf070510b4ffb928c9b8ee473.

Contracts:

- WhitelistRegistry.sol

Description:

`whitelistLimitNew` is a storage variable and its reading cost is more than the reading `whitelistLimit_` memory variable.

```
contracts/WhitelistRegistry.sol:
    59:         function  setWhitelistLimit(uint256  whitelistLimit_)
external onlyOwner {
    60:             if  (whitelistLimit  ==  whitelistLimit_)  revert
SameWhitelistSize();
  61:         whitelistLimitNew = whitelistLimit_;
  62:         if (whitelistLimitNew > _whitelist.length()) {
  63:             _setWhitelistLimit(whitelistLimitNew);
  64:         } else {
```

```
        65:                                                         emit
WhitelistLimitDecreaseRequest(whitelistLimitNew);
  66:         }
  67:     }
```

The `whitelistLimitNew` variable from `shrinkWhitelist()` function can be cached to save gas.

```
contracts/WhitelistRegistry.sol:
  69:     function shrinkWhitelist(uint256 partition) external {
   70:             if (whitelistLimit == whitelistLimitNew) revert
SameWhitelistSize();
  71:         uint256 whitelistLength = _whitelist.length();
  72:         if (whitelistLimitNew < whitelistLength) {
  73:             unchecked {
  74:                 for (uint256 i = 0; i < whitelistLength; ) {
  75:                     address curWhitelisted = _whitelist.at(i);
   76:                         if (token.balanceOf(curWhitelisted) <=
partition) {
  77:                         _removeFromWhitelist(curWhitelisted);
  78:                         whitelistLength--;
  79:                     } else {
  80:                         i++;
  81:                     }
  82:                 }
  83:             }
```

```
  84:                 if (whitelistLength != whitelistLimitNew) revert
WrongPartition();
  85:             }
  86:             _setWhitelistLimit(whitelistLimitNew);
  87:         }
```

`resolverThreshold` is a storage variable that reads in a loop and can be cached to save gas.

contracts/WhitelistRegistry.sol:

```
  118:     function clean() external {
  119:         uint256 whitelistLength = _whitelist.length();
  120:         unchecked {
  121:             for (uint256 i = 0; i < whitelistLength; ) {
  122:                 address curWhitelisted = _whitelist.at(i);
  123:                     if (token.votingPowerOf(curWhitelisted) <
resolverThreshold) {
  124:                     _removeFromWhitelist(curWhitelisted);
  125:                     whitelistLength--;
  126:                 } else {
  127:                     i++;
  128:                 }
  129:             }
  130:         }
  131:     }
```

Remediation:

Consider caching `resolverThreshold` and `whitelistLimitNew` for `clean()` and `shrinkWhitelist()` functions and use the `whitelistLimit_` variable for `setWhitelistLimit()`.

4.4. Non optimal decrement/increment

Risk Level: Info

Status: The vulnerability has been acknowledged by the Customer, no fixes will be deployed.

Contracts:

- WhitelistRegistry.sol

References:

- https://github.com/byterocket/c4-common-issues/blob/main/0-Gas-Optimizations.md/#g012---use-prefix-increment-instead-of-postfix-increment-if-possible

Description:

The difference between the prefix increment and postfix increment expression lies in the return value of the expression. The prefix increment expression (`++i`) returns the *updated* value after it's incremented. The postfix increment expression (`i++`) returns the original value. The prefix increment expression is cheaper in terms of gas. Consider using the prefix increment expression whenever the return value is not needed.

There are the following occurrences:

```
contracts/WhitelistRegistry.sol:
   78:     whitelistLength--;
   80:     i++;
  125:     whitelistLength--;
  127:     i++;
```

Remediation:

An example of a not optimized code:

```
for (uint i = 1; i <= len; i++) {
```

Consider the following example to save gas:

```
for (uint i = 1; i <= len; ++i) {
```

4.5. Integer overflow (not exploitable)

Risk Level: Info

Status: The vulnerability has been acknowledged by the Customer, no fixes will be deployed.

Contracts:

- FeeBank.sol

Description:

There is an unchecked block in the `_settleOrder()` function that calculates the `resolverFee` variable which stores the total amount of fees that will be charged from the resolver.

```
contracts/Settlement.sol:
     155:  unchecked {
     156:        resolverFee    +=    fusionDetails.resolverFee()    *
_ORDER_FEE_BASE_POINTS;
     157:        suffixLength = DynamicSuffix._STATIC_DATA_SIZE +
     158:              tokensAndAmounts.length +
     159:              (address(token) != address(0) ? 0x60 : 0x20);
     160:  }
```

It's not possible to have an overflow at the current state of the contract because `resolverFee` is uint256 type, `fusionDetails.resolverFee()` is bytes4, and `_ORDER_FEE_BASE_POINTS` is equal to 1e15, so it's not possible to have an overflow here.

In the case in the future any of this type will change it can create a possibility of overflow.

Remediation: Cover this scenario with tests.

4.6. Unnecessary ADD operator

Risk Level: Info

Status: Fixed in the commit f8bc44868a6e6dfaf070510b4ffb928c9b8ee473.

Contracts:

- Settlement.sol

Description:

There is an unnecessary `add` operator.

```
contracts/Settlement.sol:
175:                mstore(add(offset, 0x00), resolver)
```

Remediation:

Consider removing the unnecessary operator.

5. Appendix

5.1. About us

The Decurity (former DeFiSecurity.io) team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.