SALUS SECURITY

DEC 2025

CODE SECURITY ASSESSMENT

IRO GLOBAL

Overview

Project Summary

- Name: IROGlobal - IRO contract
- Platform: Xone
- Language: Solidity
- Repository:
  - https://github.com/iroglobal/iro-contract
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	IROGlobal - IRO contract
Version	v2
Type	Solidity
Dates	Dec 03 2025
Logs	Dec 02 2025; Dec 03 2025

Vulnerability Summary

Total High-Severity issues	3
Total Medium-Severity issues	4
Total Low-Severity issues	4
Total informational issues	5
Total	16

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Anti-bot lastBuyBlock logic can be abused to block user's transaction	High	Business Logic	Resolved
2	Pre-create pair dos on IRO subscribe	High	Business Logic	Resolved
3	Lock-in period reset attack all time	High	Business Logic	Resolved
4	Missing non-zero validation for initPrice in createProject	Medium	Data Validation	Resolved
5	Integer division in getLPValueForUSDTValue will cause transferAward reverts	Medium	Numerics	Resolved
6	Lock-in period reset attack on subscribe	Medium	Business Logic	Resolved
7	Missing Test Suite	Medium	Testing	Resolved
8	Missing slippage check	Low	Business Logic	Acknowledged
9	Naive one-sided liquidity in pledge traps user funds in the contract	Low	Business Logic	Acknowledged
10	Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()	Low	Risky External Calls	Acknowledged
11	Missing events for functions that change critical state	Low	Logging	Resolved
12	Misspelled variable name may cause confusion	Informational	Code Quality	Resolved
13	File and contract names mismatch	Informational	Configuration	Acknowledged
14	Inconsistent license	Informational	Configuration	Resolved
15	Interface should be placed in a separate file	Informational	Code Quality	Acknowledged
16	Gas optimization suggestions	Informational	Gas Optimization	Acknowledged

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Anti-bot lastBuyBlock logic can be abused to block user's transaction	
Severity: High	Category: Business Logic
Target: - contracts/IRO.sol	

Description

The `_transfer()` function in the IROToken contract uses `lastBuyBlock` to prevent users from selling in the same block after they buy from the pair.

contracts/IRO.sol:L385-L427

```
function _transfer(address sender, address to, uint256 amount) internal {
        _updateDayProduce();
        address pair = getPair();
        if (sender != pair && to != pair && !isTeamFee) {
            _updateDaySwap();
        }
        if (sender == pair || lastBuyBlock[sender] == block.number) {
            lastBuyBlock[to] = block.number;
        }
        ...

        if (
            (to == pair || sender == pair) && !noSwapFee && to != getAutoBuyFeeReceiver()
                && (
                    !IIROOwner(ownerAddr).isExcludedFromFee(address(this), sender)
                        && !IIROOwner(ownerAddr).isExcludedFromFee(address(this), to)
                )
        ) {
            if (block.number <= lastBuyBlock[sender]) revert Err();
            ...
}
```

This enables the following griefing pattern within a single block:

1. Attacker first buys from the pair into their own address attacker:
   - `_transfer(pair, attacker, ...)`
   - Sets `lastBuyBlock[attacker] = currentBlock`.
2. In the same block, attacker sends a small amount of tokens to the victim:
   - `_transfer(attacker, victim, ...)`
   - `lastBuyBlock[attacker] == currentBlock` is true, so `lastBuyBlock[victim] = currentBlock`.
3. When the victim's pending transaction in that block tries to sell to the pair:
   - `_transfer(victim, pair, ...)`

- The swap fee/trading branch is entered and the anti-bot check runs:
  `` `if (block.number <= lastBuyBlock[sender]) revert Err();` ``
- Since `` `sender = victim` `` and `` `lastBuyBlock[victim] = currentBlock` ``, the condition is true and the sell reverts with `` `Err()` ``.

A dust transfer in the same block is sufficient to cause all victim sells in that block to revert. This attack effectively turns the anti-bot logic into a block-level DoS on selling for arbitrary users.

Recommendation

Restrict the `` `lastBuyBlock` `` marking to direct buys from the pair only.

Status

The team has resolved this issue in commit 785b181.

2. Pre-create pair dos on IRO subscribe

Severity: High	Category: Business Logic
Target: - contracts/IRO.sol	

Description

The `subscribe()` function in the IROToken contract relies on the `_toAddLP()` internal logic to add initial liquidity for users. After liquidity is correctly added, `tokenPrice` in the following calculation will not be zero, avoiding an EVM division error.

contracts/IRO.sol:L225-L270

```
function subscribe(address _user, uint256 amount) external payable override lock {
    ...
    (,, uint256 tokenPrice) = getTokenPrice();

    uint256 tokenAmount = (tokenTKA * 10 ** 18) / tokenPrice;
    ...
}
```

According to the `getTokenPrice()` function, `newPrice` will only be accurately calculated if `pairToken` is greater than zero. When the pool exists but `reserve0` and `reserve1` are both zero, `newPrice` returns zero, which subsequently triggers a division by zero error. Before the first user subscribes, an attacker can invoke `UniswapV2Factory::createpair()` to create the associated pool, which results in a DoS condition for the IRO contract.

contracts/IRO.sol:L311-L321

```
function getTokenPrice() public view override returns (uint256 pairToken, uint256
pairTKA, uint256 newPrice) {
    address pair = getPair();
    if (pair == address(0)) return (0, 0, price);
    (uint112 reserve0, uint112 reserve1,) = IUniswapV2Pair(pair).getReserves();
    address token0 = IUniswapV2Pair(pair).token0();
    bool isToken0TKA = (token0 == TKA);
    pairTKA = isToken0TKA ? reserve0 : reserve1;
    pairToken = isToken0TKA ? reserve1 : reserve0;
    if (pairToken > 0) {
        newPrice = (uint256(pairTKA) * 1e18) / pairToken;
    }
}
```

Proof of Concept:

```
function test_DOS() public {
    deal(USDT, ROOT_BROKER, 1000 ether);
    //create project
    vm.startPrank(ROOT_BROKER);
    IERC20(USDT).approve(address(factory), 1000 ether);
    factory.createProject(
    1,
    1000 ether,
    1 ether,
    "TEST",
    [uint256(500 ether), uint256(1000 ether)],
    USDT,
    30,
    new TokenWhiteListQuota[](0));
    vm.stopPrank();
    //attacker create associated pool
    address testToken = factory.projectIDToken(1);
    deal(USDT, address(0x4), 1000 ether);
    vm.startPrank(address(0x4));
    address pair = IUniswapV2Factory(_UNISWAP_V2_FACTORY).createPair(USDT, testToken);
    vm.stopPrank();

    //user first call subscribe
    vm.startPrank(address(user));
    deal(USDT, address(user), 1000 ether);
    IERC20(USDT).approve(address(testToken), 1000 ether);
    IROToken(payable(testToken)).subscribe(address(user), 100 ether);

}
```

Recommendation

Add additional checks to ensure `getTokenPrice()` does not return zero.

Status

The team has resolved this issue in commit 785b181.

3. Lock-in period reset attack all time

Severity: High	Category: Business Logic
Target: - contracts/IRO.sol	

Description

When a user calls `` `removePledge()` ``, the protocol uses the `` `userProperty.lastTimestamp` `` to calculate position duration, which determines the applicable fee rate.

contracts/IRO.sol:L640-L663

```
function removePledge() external override {
    …
    uint256 keepDays = (block.timestamp - userProperty.lastTimestamp) / ONEDAY;
    uint256 rate = IIROOwner(IROToken(token).ownerAddr()).getTaxRate(keepDays,
pledgeDays);
    uint256 feeAmount = (pledgeNum * rate) / 100;
    …
}
```

For IRO projects with `` `pledgeDays` `` < 90 days, a flaw exists in the IROPool contract's `` `_transferAward()` `` logic. When user A receives equity transfers from user B, the contract resets user A's `` `lastTimestamp` `` regardless of the transfer amount. This allows an attacker to grief users approaching their unlock period by repeatedly sending dust amounts of equity, continuously resetting their lock period and indefinitely preventing them from withdrawing principal at favorable tax rates. The victim is unable to achieve maturity conditions, forcing withdrawal at the highest penalty tier.

contracts/IRO.sol:L601-L629

```
function _transferAward(address from, address to, uint256 TKAAmount) internal {
    …
    if (toProperty.lastTimestamp == 0 || pledgeDays < 90) {
        toProperty.lastTimestamp = block.timestamp;
    }
    ..
}

function transferAward(address to, uint256 TKAAmount) external override {
    _transferAward(msg.sender, to, TKAAmount);
}
```

Proof of Concept:

```
function test_reset_user_lastTimestamp_by_transferAward() public {
    deal(USDT, ROOT_BROKER, 1000 ether);
    vm.startPrank(ROOT_BROKER);
    IERC20(USDT).approve(address(factory), 1000 ether);
    address pool = factory.createProject(
    1,
    1000 ether,
    1 ether,
    "TEST",
    [uint256(500 ether), uint256(1000 ether)],
    USDT,
    30,
    new TokenWhiteListQuota[](0));
    vm.stopPrank();
    address testToken = factory.projectIDToken(1);
    vm.startPrank(address(0x3));
    deal(USDT, address(0x3), 10000 ether);
    vm.deal(address(0x3), 10 ether);
    IERC20(USDT).approve(address(testToken), 10000 ether);
    IROToken(payable(testToken)).subscribe(address(0x3), 100 ether);
    (,,,,uint256 lastTimestamp,) = IROPool(payable(pool)).getUserInfo(address(0x3));
    console.log("0x3 user lastTimestamp before transferAward ", lastTimestamp);
    vm.stopPrank();
    //increase block.timestamp
    vm.warp(block.timestamp + 1 days);

    vm.startPrank(address(0x4));
    deal(USDT, address(0x4), 10000 ether);
    IERC20(USDT).approve(address(testToken), 10000 ether);
    IROToken(payable(testToken)).subscribe(address(0x4), 100 ether);
    IROPool(payable(pool)).transferAward(address(0x3), 2 ether);
    (,,,,uint256 lastTimestamp2,) = IROPool(payable(pool)).getUserInfo(address(0x3));
    console.log("0x3 user lastTimestamp after transferAward", lastTimestamp2);
    vm.stopPrank();
}
```

Recommendation

Redesign the `transferAward` logic.

Status

The team has resolved this issue in commit 785b181.

4. Missing non-zero validation for initPrice in createProject

Severity: Medium	Category: Data Validation
Target: - contracts/IRO.sol	

Description

In `IROFactory.createProject()`, the factory does not validate that `_initPrice` is non-zero before passing it to `IROPool.initialize()`. As a result, a project can be created with `initPrice == 0`. In `IROPool.initialize()`, this value is stored as `lastPrice`. Later, when `IROPool.addProduceNum()` is called via `IROToken._updateDayProduce()`, the function computes the daily amplitude based on price appreciation:

contracts/IRO.sol:L708-L739

```
function addProduceNum() external override {
        if (msg.sender != token) return;
        (uint256 pairToken,, uint256 newPrice) = IROToken(token).getTokenPrice();
        if (newPrice == 0) return;

        uint256 amplitude;
        if (newPrice > lastPrice) {
            uint256 _increaseRate = (((newPrice - lastPrice) * 10000) / lastPrice);
            increaseRateMap[block.timestamp / ONEDAY].rate = _increaseRate;
            amplitude = _increaseRate + 200;
            amplitude = amplitude > 5000 ? 5000 : amplitude;
        } else {
            amplitude = 200;
        }
       ...
    }
```

If `lastPrice` is zero and there is any non-zero `newPrice` once liquidity exists, this division will revert due to a division by zero. Because `addProduceNum()` is invoked indirectly from `_updateDayProduce()` inside `_approve()` and `_transfer()`, a misconfigured project with `_initPrice == 0` can cause routine token operations (`approvals/transfers`) to revert, effectively putting the token into a semi-frozen state. At the same time, the pool's reward emission logic cannot progress, meaning LP rewards and share accounting for that project are effectively broken.

Recommendation

Add `initPrice` validation in `createProject()`.

Status

The team has resolved this issue in commit 785b181.

5. Integer division in getLPValueForUSDTValue will cause transferAward reverts	
Severity: Medium	Category: Numerics
Target: - contracts/IRO.sol	

Description

The `getLPValueForUSDTValue()` used to convert a target `TKA` value into an equivalent `LP` amount suffers from severe precision loss due to integer division order. For many realistic inputs it returns 0, which then causes `transferAward()/transferAwards()` to revert.

contracts/IRO.sol:L591-L597

```
function getLPValueForUSDTValue(uint256 targetTKAValue) public view returns (uint256
valueInTKA) {
    address pair = getPair();
    (, uint256 pairTKA,) = IROToken(token).getTokenPrice();
    uint256 totalSupply = IUniswapV2Pair(pair).totalSupply();
    uint256 lpUintPrice = ((pairTKA * 10 ** 18) / totalSupply) * 2;
    valueInTKA = (targetTKAValue / lpUintPrice) * 10 ** 18;
}
```

Recommendation

Fix the conversion formula to avoid premature truncation by multiplying before dividing.

Status

The team has resolved this issue in commit 785b181.

6. Lock-in period reset attack on subscribe	
Severity: Medium	Category: Business Logic
Target: - contracts/IRO.sol	

Description

The `subscribe()` function in the IROToken contract allows users to subscribe on behalf of others (address `_user`). Whenever a user's new staking action is triggered, the IROPool contract forcibly resets the user's `lastTimestamp`.

contracts/IRO.sol:L225-L270,L360-L365,L565-L582

```
function subscribe(address _user, uint256 amount) external payable override lock {
    ...

    _toAddLP(_user, TKAAmount, tokenTKA, tokenAmount);
    ..
}
function _toAddLP(address pledgeTo, uint256 share, uint256 TKAAmt, uint256 tokenAmt)
internal {
    ....
    IIROPool(POOL).pledge(pledgeTo, lp, share);
}
function pledge(address user, uint256 amount, uint256 TKAAmount) external override {
    …
    if (userProperty.lastTimestamp == 0 || pledgeDays < 90) {
        userProperty.lastTimestamp = block.timestamp;
    }

    totalShare += share;
    userProperty.pledgeNum += amount;
    emit Pledge(user, amount, share);
}
```

Proof of Concept:

```
function test_reset_user_lastTimestamp_by_subscribe() public {
    deal(USDT, ROOT_BROKER, 1000 ether);
    vm.startPrank(ROOT_BROKER);
    IERC20(USDT).approve(address(factory), 1000 ether);
    address pool = factory.createProject(
    1,
    1000 ether,
    1 ether,
    "TEST",
    [uint256(500 ether), uint256(1000 ether)],
    USDT,
    30,
    new TokenWhiteListQuota[](0));
    vm.stopPrank();
    address testToken = factory.projectIDToken(1);
    vm.startPrank(address(0x3));
    deal(USDT, address(0x3), 10000 ether);
    vm.deal(address(0x3), 10 ether);
    IERC20(USDT).approve(address(testToken), 10000 ether);
    IROToken(payable(testToken)).subscribe(address(0x3), 100 ether);
    (,,,,uint256 lastTimestamp,) = IROPool(payable(pool)).getUserInfo(address(0x3));
    console.log("block.timestamp", lastTimestamp);
    vm.stopPrank();

    vm.warp(block.timestamp + 1 days);

    vm.startPrank(address(0x4));
    deal(USDT, address(0x4), 10000 ether);
    IERC20(USDT).approve(address(testToken), 10000 ether);
    IROToken(payable(testToken)).subscribe(address(0x3), 100); //for 100 ether it's very
small
    (,,,,uint256 lastTimestamp2,) = IROPool(payable(pool)).getUserInfo(address(0x3));
    console.log("block.timestamp", lastTimestamp2);
    vm.stopPrank();
}
```

Recommendation

Remove the ability to subscribe on behalf of others.

Status

The team has resolved this issue in commit [785b181](785b181).

7. Missing Test Suite	
Severity: Medium	Category: Testing
Target: - All	

Description

A test suite to confirm the proper operation of the deployed contracts and crucial features like registering and admin role assignment is absent from the repository. Contract security and dependability are at risk because defects or regressions might be undiscovered. Test scripts with filenames ending in.ts are often located in the test/ directory in Harhat-based projects.

Recommendation

Think about leveraging Harhat to create a thorough test suite that covers essential functions. Ensuring contract correctness and promoting safe continuous development require a strong test suite.

Status

The team has resolved this issue in commit 1a6cbfa.

8. Missing slippage check	
Severity: Low	Category: Business Logic
Target: - contracts/IRO.sol	

Description

The protocol uses Uniswap V2 router's `swapExactETHForTokens()` and `swapExactTokensForTokens()` with `amountOutMin` hardcoded to 1, which is effectively equivalent to no slippage protection. This allows trades to suffer excessive slippage and exposes them to price manipulation, such as sandwich/front-running attacks, especially since these swaps are executed using protocol-owned funds in `_updateDaySwap()`, `subscribe()`, and `pledge()`.

contracts/IRO.sol:L335-347

```
function _toSwapToken(uint256 TKAAmount, address to) internal returns (uint256[] memory
amounts) {
   address[] memory path = new address[](2);
   path[0] = TKA;
   path[1] = address(this);
   if (isWXOC) {
       amounts = IUniswapV2Router(UNISWAP_V2_ROUTER).swapExactETHForTokens{value:
TKAAmount}(1, path, to, block.timestamp);
   } else {
       amounts =
           IUniswapV2Router(UNISWAP_V2_ROUTER).swapExactTokensForTokens(TKAAmount, 1,
path, to, block.timestamp);
   }
}
```

Recommendation

Implement a slippage check.

Status

This issue has been acknowledged by the team. The team states that the tax rate mechanism makes sandwich attacks very difficult to execute.

9. Naive one-sided liquidity in pledge traps user funds in the contract	
Severity: Low	Category: Business Logic
Target: - contracts/IRO.sol	

Description

The `IROToken.pledge()` function uses a naive `amount/2` split to perform a one-sided add-liquidity operation on a Uniswap V2–style pool. After swapping half of the user's `TKA` into the token and calling `addLiquidity()`, any surplus `TKA/token` caused by the post-swap price shift is not refunded to the user and instead remains in the contract (and token leftovers are even burned):

contracts/IRO.sol:L292-L308

```
function pledge(address _user, uint256 amount) external payable lock {
    if (isContract(msg.sender) || isContract(_user)) revert notContract();
    if (!isOpen) revert isNoOpened();
    verifyAmount(amount);
    noSwapFee = true;
    uint256 buyAmount = amount / 2;
    uint256[] memory amounts = _toSwapToken(buyAmount, msg.sender);
    uint256 amountOut = (amounts[amounts.length - 1]);

    TransferHelper.safeTransferFrom(address(this), msg.sender, address(this), amountOut);

    _toAddLP(_user, amount, amount - buyAmount, amountOut);
    if (balances[address(this)] > 0) {
        _transfer(address(this), address(0), balances[address(this)]);
    }
    noSwapFee = false;
}
```

Because swapping `amount/2` always moves the price, the remaining `TKA` and swapped token rarely match the pool ratio exactly. `addLiquidity()` will only consume what fits the current ratio and leave residual token. In this design, token leftovers are burned. Over time this causes systematic funds loss from users.

Recommendation

Replace the naive `amount/2` split with an optimal one-sided liquidity formula based on current reserves.

Status

This issue has been acknowledged by the team. The team states that this loss is within acceptable parameters.

10. Use safeTransfer()/safeTransferFrom() instead of transfer()/transferFrom()	
Severity: Low	Category: Risky External Calls
Target: - contracts/IRO.sol - contracts/Vault.sol	

Description

Tokens not compliant with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

```
Callers MUST handle false from returns (bool success). Callers MUST NOT assume that
false is never returned!
```

Recommendation

Consider using the SafeERC20 library implementation from OpenZeppelin and call safeTransfer or safeTransferFrom when transferring ERC20 tokens.

Status

This issue has been acknowledged by the team.

11. Missing events for functions that change critical state	
Severity: Low	Category: Logging
Target: - contracts/IRO.sol - contracts/Vault.sol - contracts/Owner.sol	

Description

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes that allow users to evaluate them. Missing events do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in protocol users.

In the `IRO` contract, events are lacking in the privileged setter functions (e.g.`setOwnerContract`).

In the `Owner` contract, events are lacking in the privileged setter functions (e.g.`setLevelFee`).

Recommendation

It is recommended to emit events for critical state changes.

Status

The team has resolved this issue in commit 785b181.

2.3 Informational Findings

12. Misspelled variable name may cause confusion	
Severity: Informational	Category: Code Quality
Target: - contracts/IRO.sol	

Description

The contract declares a public state variable with an incorrect spelling:

contracts/IRO.sol:L108

```
uint256 public lastSwapBlcokNumber;
```

Recommendation

Correct the variable name to `` `lastSwapBlockNumber` `` for clarity and consistency.

Status

The team has resolved this issue in commit 785b181.

13. File and contract names mismatch	
Severity: Informational	Category: Configuration
Target: - contracts/IRO.sol	

Description

Some files in the codebase have names that do not match the contract defined within. For example, the file `IRO` contains a contract named `IROPool`, `IROFactory` and `IROSellFeeContract`. Such mismatches can create confusion for developers and auditors, reducing code readability and maintainability.

Recommendation

It is recommended to rename the file to match the contract name.

Status

This issue has been acknowledged by the team.

14. Inconsistent license	
Severity: Informational	Category: Configuration
Target: - All	

Description

Some files use a different license declaration than the rest of the codebase. This inconsistency creates uncertainty around usage, distribution, and compliance, potentially impacting open-source compliance, collaboration, and future integrations.

Recommendation

It is recommended to align all license declarations in the codebase with the overall project license (e.g., GPL-3.0 or MIT) to reduce uncertainty in usage and distribution and improve compliance and maintainability.

Status

The team has resolved this issue in commit 785b181.

15. Interface should be placed in a separate file	
Severity: Informational	Category: Code Quality
Target: - contracts/IRO.sol - contracts/Owner.sol	

Description

The `` `IUniswapV2Factory` ``, `` `IUniswapV2Router` ``, `` `IUniswapV2Pair` ``, `` `IIROToken` `` and `` `IIROPool` `` interface are currently declared together in `` `IRO.sol` ``.

The `` `IOrganization` ``, `` `IROStake` ``, `` `IIROOwner` `` and `` `IIROFactory` `` interfaces are currently declared together in `` `Owner.sol` ``.

Combining an interface and a contract in the same file increases code complexity and reduces readability, making it harder for developers and auditors to navigate, understand, and maintain the codebase.

Recommendation

It is recommended to separate the <InterfaceName> interface into its own dedicated file. Doing so improves the clarity, organization, and maintainability of the codebase, while aligning the project with standard Solidity best practices.

Status

This issue has been acknowledged by the team.

16. Gas optimization suggestions	
Severity: Informational	Category: Gas Optimization
Target: - contracts/IRO.sol - contracts/Owner.sol	

Description

Memory reading saves more gas than storage reading multiple times when the state is not changed. So caching the storage variables in memory and using the memory instead of storage reading is effective.

contracts/Owner.sol:L300, L315,L350,L447

```
for (uint256 i = 0; i < excludedFromFeeArr[token].length; i++) {
for (uint256 i = 0; i < _whiteList.length; i++) {
for (uint256 i = 0; i < tokenWhiteListArr[token].length; i++) {
for (uint256 i = 0; i < brokerGroup.length; i++) {
```

Recommendation

Consider using the above suggestions to save gas.

Status

This issue has been acknowledged by the team.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit 4dd109e::

File	SHA-1 hash
Owner.sol	994b6e12fe970f0054e25a91b9005f71cf393c57
Vault.sol	4d6b175fa7039eb7f64d573afe311fdbe6ab0356
IRO.sol	007c11c8ff1e35ec316432e2702936d928765578
organization.sol	759e725213b98c22f55e20c3fdf423b0dde912d3