



Fundable Audit Report

Prepared by: 0xzerpa

2025-07-24

Table of Contents

- [Table of Contents](#)
- [Protocol Summary](#)
- [Disclaimer](#)
- [Risk Classification](#)
- [Audit Details](#)
 - [Scope](#)
 - [Roles](#)
- [Executive Summary](#)
 - [Issues found](#)
- [Findings](#)
- [High](#)
- [Medium](#)
- [Low](#)
- [Informational](#)
- [Gas](#)

Audit Summary

Fundable is a decentralized finance protocol built on StarkNet that combines token distribution and payment streaming capabilities. Fundable enables both bulk token distributions and continuous, real-time token streaming with advanced management features.

Oxzerpa is an independent Smart Contract Security Researcher with background on web and defi development and he is performing this audit.

Disclaimer

Oxzerpa makes every effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. This security audit is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Cairo implementation of the contracts.

This audit IS NOT a guarantee that the protocol is 100% free of bugs or vulnerabilities.

Risk Classification

		Impact		
		High	Medium	Low
		H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

This audit focused on the payment streaming functionality of the protocol. Our review specifically verified the correct operation of the following characteristics:

- Linear token streaming
- Stream management
- Partial withdrawals
- Stream status tracking

Additionally, we tried to assessed any potential vulnerabilities arising from the logic implementation.

The audit was performed on the commit hash
6ecc05e064619ff714759e57bb115aaee82b84aa.

Scope

- `src/payment_stream.cairo`

Roles

- contract owner
- stream sender
- stream receiver
- delegatee
- approved
- fee recipient

Executive Summary

Issues found

- **[M-1]** Unchecked transfer/transferFrom Return Values Lead to State-Balance Inconsistencies and possible thief of funds of other streams
- **[M-1]** Re-pausing a Stream Resets `paused_rates` to Zero, Causing a DoS (Denial of Service) for Future Withdrawals
- **[L-1]** `cancel` function is not checking the `stream.cancelable` flag, so it is being executing regardless such flag value
- **[NC-1]** `_refund` Function Lacks Input Validation, Posing Future Risk
- **[NC-2]** No custom error on assertion in `calculate_stream_rate` function
- **[GAS-1]** Redundant Assertions in `_deposit`
- **[GAS-2]** Duplicate `stream_metrics.last_activity` Update in `_deposit`
- **[GAS-3]** Unnecessary Stream Existence Check in `pause`

Findings

[M-1] Unsafe transfer/transferFrom calls lead to State-Balance Inconsistencies and possible thief of funds of other streams

Many ERC20 implementations return `false` on `transfer` or `transferFrom` failure instead of reverting. If the protocol doesn't assert this boolean return, the execution flow continues, updating internal storage variables despite a failed token transfer. This desynchronizes the contract's internal state from the actual token balances. For example: Consider `create_stream` which invokes `_deposit` and subsequently `transferFrom`. If `transferFrom` returns `false`, the stream is still erroneously registered. Later, attempts to withdraw from this stream will fail due to insufficient actual funds. In more severe cases, this could lead to funds being incorrectly debited from other streams balances.

The above example is just one of the many unexpected scenarios that can be caused due to this vulnerability.

Mitigation

An intuitive solution will be to assert the return value of all the calls to `transfer` and `transferFrom` functions, for example:

```
let success =
token_dispatcher.transfer_from(get_caller_address(),get_contract_address(),
(), amount);

assert(success, CUSTOM_ERROR_FOR_THIS_CASE);
```

But this will not work with weird ERC20 tokens that do not return any boolean value (ex. USDT) so a better approach will be to cache the balance of the contract pre and post transfer and then evaluate the difference between both of them to verify successful transaction execution.

A perfect example of this is OpenZeppelin's safeERC20 implementation, but it can not be implemented here because there is still not a cairo version of safeERC20.

[M-2] Re-pausing a Stream Resets `paused_rates` to Zero, Causing a DoS (Denial of Service) for Future Withdrawals

The pause function is designed to save a stream's `rate_per_second` into `paused_rates` before setting the active stream rate to zero. However, if `pause` is invoked on an already paused stream, `paused_rates` will be overwritten with the stream's current `rate_per_second` – which is 0.

When the `restart` function is subsequently called, it will restore `stream.rate_per_second` from this now-zero `paused_rates` value. Consequently, the stream's rate will become 0 permanently. Since the withdrawable amount for a stream is calculated using `stream.rate_per_second` as a multiplier, all future withdrawal attempts will result in a zero amount, effectively locking funds within the stream.

Mitigation Verify that the value registered in `paused_rates` for a stream is zero, to then proceed with the update:

In line 921

```
+     if self.paused_rates.read(stream_id) == 0 {  
  
        // Store the current rate before pausing  
  
        self.paused_rates.write(stream_id, stream.rate_per_second);  
  
    }  
}
```

[L-1] `cancel` function is not checking the `stream.cancelable` flag, so it is being executing regardless such flag value

mitigation Add an asset to check for the cancelable boolean of the stream to be true.

In line 946

```
// Retrieve the stream  
  
let mut stream = self.streams.read(stream_id);  
  
+ assert(stream.cancelable, 'CREATE_CUSTOM_ERROR_FOR_THIS');  
  
let stream_balance = stream.balance;
```

Informational/Non Critical (NC)

[NC-1] `_refund` Function Lacks Input Validation, Posing Future Risk

The internal `_refund` function currently allows a stream sender to specify an arbitrary amount to be refunded, without any checks to ensure this amount doesn't exceed the actual refundable balance. While `_refund` isn't called by any public function right now, its presence introduces a potential vulnerability if it's integrated into future public-facing logic. An attacker could exploit this to withdraw more funds than legitimately available. To prevent future exploits, it's highly recommended to either remove or comment out the `_refund` function, or implement robust input validation to assert that the amount parameter does not exceed the sender's available refund balance.

[NC-2] No custom error on assertion in `calculate_stream_rate` function

The assertion `assert(total_amount <= max_safe_amount, 'Amount too large for scaling');` in line 293, is returning plain text instead of a custom error.

Gas

[GAS-1] Redundant Assertions in `_deposit`

The `_deposit` function includes two unnecessary assert statements:
`assert(deposit_diff >= amount, OVERDEPOSIT);` and `assert(stream.status != StreamStatus::Canceled, STREAM_CANCELED);`. Both stream creation and deposits occur within a single transaction, rendering these checks superfluous. They likely remain from a previous design where these actions were separate steps. Removing them will save gas.

[GAS-2] Duplicate `stream_metrics.last_activity` Update in `_deposit`

The `_deposit` function redundantly updates `stream_metrics.last_activity`. This value is already set to the current timestamp by `_create_stream`, which is called immediately before `_deposit`. This line can be safely removed to optimize gas usage.

[GAS-3] Unnecessary Stream Existence Check in `pause`

The `pause` function performs a redundant check for stream existence. It first calls `assert_stream_sender_access`, which internally invokes `self.assert_stream_exists(stream_id)`. A few lines later, `self.assert_stream_exists(stream_id)` is called again directly. The second check is unnecessary and can be removed to reduce gas costs.