Sablier Stream Audit Report

Prepared by: - MOHAMED IBRAHIIM

Table of Contents

Risk Classification

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

Audit Details

Scope

All Contracts in `src` are in scope except `interfaces`.

```
src/
|-- FlowNFTDescriptor.sol
|-- SablierFlow.sol
|-- abstracts/
|   |-- Adminable.sol
|   |-- Batch.sol
|   |-- NoDelegateCall.sol
|   |-- SablierFlowBase.sol
|-- libraries/
|   |-- Errors.sol
|   |-- Helpers.sol
|-- types/
    |-- DataTypes.sol
```

Protocol Summary

How Flow works

One can create a flow stream without any upfront deposit, so the initial stream balance begins at zero. The sender can later deposit any amount into the stream at any time. To improve the experience, a `createAndDeposit` function has also been implemented to allow both create and deposit in a single transaction.

One can also start a stream without setting an rps. If rps is set to non-zero at the beginning, it begins streaming as soon as the transaction is confirmed on the blockchain. These streams have no end date, but it allows the sender to pause it or void it at a later date.

A stream is represented by a struct, which can be found in `DataTypes.sol`.

The debt is tracked using `snapshotDebtScaled` and `snapshotTime`. At snapshot, the following events are taking place:

1. `snapshotDebtScaled` is incremented by `ongoingDebtScaled` where `ongoingDebtScaled = rps * (block.timestamp - snapshotTime)`
2. `snapshotTime` is updated to `block.timestamp`

The recipient can withdraw the streamed amount at any point. However, if there aren't sufficient funds, the recipient can only withdraw the available balance.

Abbreviations

Terms	Abbreviations
Block Timestamp	now
Covered Debt	cd
Ongoing Debt	od
Rate per second	rps
Refundable Amount	ra
Scale Factor	sf
Snapshot Debt	sd
Snapshot Time	st
Stream Balance	bal

Terms	Abbreviations
Time elapsed since snapshot	elt
Total Debt	td
Uncovered Debt	ud

Invariants

1. For any stream, st <= now
2. For a given token:
    - Sum stream balances + protocol revenue = aggregate balance
    - token.balanceOf(SablierFlow) >= Sum stream balances + flow.protocolRevenue(token)
    - Sum stream balances = Sum deposited amount - Sum refunded amount - Sum withdrawn amount
3. For a given token, token.balanceOf(SablierFlow) >= flow.aggregateBalance(token)
4. Snapshot time should never decrease
5. For any stream, if ud > 0 => cd = bal
6. If rps > 0 and no deposits are made => d(ud)/dt >= 0
7. If rps > 0, and no withdraw is made => d(td)/dt >= 0
8. For any stream, sum of deposited amounts >= sum of withdrawn amounts + sum of refunded
9. Sum of all deposited amounts >= sum of all withdrawn amounts + sum of all refunded
10. Next stream id = current stream id + 1
11. If ud = 0 => cd = td
12. bal = ra + cd
13. For any non-voided stream, if rps > 0 => isPaused = false and Flow.Status is either STREAMING_SOLVENT or STREAMING_INSOLVENT
14. For any non-voided stream, if rps = 0 => isPaused = true and Flow.Status is either PAUSED_SOLVENT or PAUSED_INSOLVENT
15. If isPaused = true => rps = 0
16. If isVoided = true => isPaused = true and ud = 0

17. If isVoided = false => expected amount streamed >= td + amount withdrawn and expected amount streamed - (td + amount withdrawn) <= 10

Limitation

- ERC-20 tokens with decimals higher than 18 are not supported.

Core components

1. Ongoing debt The ongoing debt (od) is the debt accrued since the last snapshot. It is defined as the rate per second (rps) multiplied by the time elapsed since the snapshot time.

od = rps × elt = rps × (now - st)

2. Snapshot debt The snapshot debt (sd) is the amount that the sender owed to the recipient at the snapshot time. During a snapshot, the snapshot debt increases by the ongoing debt.

sd = sd + od

3. Total debt The total debt (td) is the total amount the sender owes to the recipient. It is calculated as the sum of the snapshot debt and the ongoing debt.

td = sd + od

4. Covered debt The part of the total debt that covered by the stream balance. This is the same as the withdrawable amount, which is an alias.

The covered debt (cd) is defined as the minimum of the total debt and the stream balance.

cd = td if td <= bal cd = bal if td > bal

5. Uncovered debt The part of the total debt that is not covered by the stream balance. This is what the sender owes to the stream.

The uncovered debt (ud) is defined as the difference between the total debt and the stream balance, applicable only when the total debt exceeds the balance.

ud = td - bal if td > bal ud = 0 if td <= bal

Together, covered debt and uncovered debt make up the total debt.

6. Refundable amount The refundable amount (ra) is the amount that can be refunded to the sender. It is defined as the difference between the stream balance and the total debt.

ra = bal - td if ud = 0 ra = 0 if ud > 0

About precision

The `rps` introduces a precision problem for tokens with fewer decimals (e.g., USDC, which has 6 decimals).

Let's consider an example: if a user wants to stream 10 USDC per day, the rps should be:

rps = 0.000115740740740740740740… (infinite decimals)

But since USDC only has 6 decimals, the rps would be limited to 0.000115, leading to 0.000115 × seconds in one day = 9.936000 USDC streamed in one day. This results in a shortfall of 0.064000 USDC per day, which is problematic.

Defining rps as 18 decimal number

In the contracts, we scale the rate per second to 18 decimals. While this doesn't completely solve the issue, it significantly minimizes it.

Using the same example (streaming 10 USDC per day), if rps has 18 decimals, the end-of-day result would be:

0.000115740740740740 × seconds in one day = 9.999999999999936000

The difference would be:

10.000000000000000000 - 9.999999999999936000 = 0.000000000000064000

This is an improvement by approximately 10^11. While not perfect, it is clearly much better as the recipient may have to wait just a bit longer to receive the full 10 USDC per day. Using the 18 decimals format would delay it by just 1 more second:

0.000115740740740740 × (seconds in one day + 1 second) = 10.000115740740677000

Currently, it's not possible to address this precision problem entirely.

Executive Summary

Issues Found

Severity Level	Number of Issues
Medium	3
Low	1
Total	4

Findings

Findings Summary

Finding ID	Description
MEDIUM	`SablierFlowBase` lacks `EIP-165` compliance for `EIP4906` interface support.
MEDIUM	`depletionTimeOf()` can return 0 incorrectly when balance equals debt.
MEDIUM	Users can escape paying fees when withdrawing and depositing via a broker.
LOW	Tokens without the `decimals()` implementation may cause errors in stream creation.

Medium

[M-1] `SablierFlowBase` Lacks `EIP-165` Compliance for `EIP4906` Interface Support

Description: The `SablierFlowBase` contract does not adhere to the `EIP4906` standard, as it fails to implement the required `supportsInterface(bytes4)` function to confirm compatibility with the `IERC4906` interface. This omission can cause integration issues for contracts that rely on `EIP-165` to verify interface support.

Impact: The lack of a proper `supportsInterface` implementation can cause significant compatibility issues. Systems that check for `EIP-165` compliance may fail to interact correctly with the `SablierFlowBase` contract. This could lead to failures in contract interactions, integrations, and potentially disrupt decentralized applications relying on `SablierFlowBase`.

Proof of Concept:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.22;
```

```
3
4  import { IERC4906 } from "@openzeppelin/contracts/interfaces/IERC4906.
       sol";
5  import { IERC165 } from "@openzeppelin/contracts/utils/introspection/
       IERC165.sol";
6  import { IERC721 } from "@openzeppelin/contracts/token/ERC721/ERC721.
       sol";
7  import { IERC721Metadata } from "@openzeppelin/contracts/token/ERC721/
       extensions/IERC721Metadata.sol";
8
9  import { Base_Test } from "./Base.t.sol";
10 import { console } from "forge-std/src/console.sol";
11
12 contract Chista0xAudit is Base_Test {
13     function test_supportsInterface_IERC165() external view {
14         bool res = flow.supportsInterface(type(IERC165).interfaceId);
15         assertEq(res, true, "EIP165 interface not supported");
16     }
17
18     function test_supportsInterface_IERC721() external view {
19         bool res = flow.supportsInterface(type(IERC721).interfaceId);
20         assertEq(res, true, "IERC721 interface not supported");
21     }
22
23     function test_supportsInterface_IERC721Metadata() external view {
24         bool res = flow.supportsInterface(type(IERC721Metadata).
               interfaceId);
25         assertEq(res, true, "IERC721Metadata interface not supported");
26     }
27
28     function test_supportsInterface_IERC4906() external view {
29         bool res = flow.supportsInterface(type(IERC4906).interfaceId);
30         assertEq(res, true, "IERC4906 interface not supported");
31     }
32 }
```

Recommended Mitigation: add `supportsInterface` function to the `SablierFlowBase` contract.

`src\abstracts\SablierFlowBase.sol`:

```
1  + import { IERC4906 } from "@openzeppelin/contracts/interfaces/IERC4906
       .sol";
2  + import { IERC165 } from "@openzeppelin/contracts/utils/introspection/
       IERC165.sol";
3
4  + function supportsInterface(bytes4 interfaceId) public view virtual
       override(ERC721, IERC165) returns (bool) {
5  +     return interfaceId == type(IERC4906).interfaceId || super.
       supportsInterface(interfaceId)}
```

[M-2] `depletionTimeOf()` can return 0 and a timestamp value when balance and debt relation didn't change

Description: The issue is somewhat similar to the issue reported in Cantina's audit report section 3.3.4* (copied below). It describes a problem that depletionTimeOf() returns 0 when it should still return the timestamp of when the totalDebt will **exceed** the balance by one token. But in this case I want to highlight the return difference of when balance is and is NOT 0. Inconsistent returns from depletionTimeOf(), depending if balance is 0 or not, even though the time and rate doesn't change. This is because the function first checks if there is no balance and returns 0. But if stream has balance > 0, then it will only return 0 if the debt is above balance + 1 MVT (Minimum Value Transferable).

This creates a scenario where if the value is withdrawn at the time when stream `balance` = `totalDebt` the `depletionTimeOf()` will shift from returning a timestamp slightly into the future to returning 0. This means that the depletion isn't correctly configured of what it considers the "depletion time" if it is the timestamp of when the total debt EXCEEDS the balance or when total debt is EQUAL to the balance.

An example of the problem is that the `depletionTimeOf()` returns:

- `rps` > 0; `balance` = 1; `totalDebt` = 1 -> `output` is a `timestamp`
- `rps` > 0; `balance` = 0; `totalDebt` = 0 -> `output` is 0

Even though in both cases the values are equal, one will provide a `timestamp` and will consider the stream not yet depleted, while in the other case it is considered depleted.

Impact: Not completely accurate `depletionTimeOf()` value. Returns 0 if `balance` is 0, but returns `timestamp` if `balance` = `totalDebt` (but their return values should match).

Proof of Concept:

```
1  function test_DepletionTimeOf_balance_discrepancy() external
       givenNotNull givenNotPaused givenBalanceNotZero {
2      console.log("Rate per second        : ", flow.getRatePerSecond(
           defaultStreamId).intoUint128());
3
4      vm.warp({ newTimestamp: block.timestamp + 47_408_000 });
5
6      console.log("total debt (6 decimals): ", flow.totalDebtOf(
           defaultStreamId));
7      console.log("balance (6 decimals)   : ", flow.getBalance(
           defaultStreamId));
8
9      console.logString("---PRE withdraw---");
10
11     console.log("---now                  : ", block.timestamp);
12     console.log("---depletion time       : ", flow.depletionTimeOf(
           defaultStreamId));
```

```
13
14      vm.stopPrank();
15      vm.prank(users.recipient);
16
17      console.logString("---Withdrawing amount = stream balance---");
18      flow.withdraw(defaultStreamId, users.recipient, flow.getBalance(
            defaultStreamId));
19
20      console.logString("---POST withdraw---");
21
22      console.log("---now                  : ", block.timestamp);
23      console.log("---depletion time       : ", flow.depletionTimeOf(
            defaultStreamId));
24  }
```

```
Logs:
  Rate per second        :  1000000000000000
  total debt (6 decimals):  50000000000
  balance (6 decimals)   :  50000000000
  ---PRE withdraw---
  ---now                 :  1777740800
  ---depletion time      :  1777740801
  ---Withdrawing amount = stream balance---
  ---POST withdraw---
  ---now                 :  1777740800
  ---depletion time      :  0
```

This discrepancy is very minor edge case, that the function can return two different outputs at the same time without changing the rates. And that the depletion time doesn't have to be reached for it to be "depleted". Which is a slight conflict in the depletionTimeOf logic.

Recommended Mitigation:

Probably the simplest solution is to simply remove the `if(balance == 0)` case and always follow the same logic path of totalDebt has to EXCEED the balance. Otherwise alter the further down logic checks to switch to checking when the totalDebt EQUALS the balance.

Change the `depletiontimeOf()` as follows:

```
- if (snapshotDebt + _ongoingDebtOf(streamId) >= balance) {
+ if (snapshotDebt + _ongoingDebtOf(streamId) > balance) {
```

[M-3] Users can escape paying fees when withdrawing and depositing via a broker.

Description: The contracts don't enforce a minimum amount to be deposited or withdrawn. This allows users to send batched deposits/withdraws for small amounts to escape paying fees by playing with the required amounts to make the computation of fees to be 0 and netAmount to be equals to the

full specified amount. Fees are charged when withdrawing (protocol fees) and when depositing via a broker (broker fee). The `Helpers.calculateAmountsFromFee()` `function` is in charged to compute the fees and the netAmount the users would withdraw/deposit.

There is not a minimum amount to deposit or withdraw, the requests are processed as long as the amount is != 0. This opens up the doors for **users to escape paying fees by depositing / withdrawing small amounts.**

- For tokens with low decimals the users can deposit/withdraw more USD value in each execution. For example, the stablecoin `GUSD`, which uses 2 decimals.
  - 1 GUSD is expressed as 1e2 (100).

For example, a protocolFee of 1% for the GUSD token would allow users to withdraw 99 unit of GUSD (0.99 USD) without paying fees.

- To demonstrate the problem, find below a coded PoC that shows how by setting `totalAmount == 99`, and the fee at 1%, the computed `feeAmount is 0`.

Create a new file under the `tests/ folder` and add the next PoC. Run it with the command `forge test --match-test test_calculateAmountsFromFee -vvvv`

- The next PoC demonstrates how the computation of fees for a small amount of GUSD allows users to skip paying fees.
  - feeAmount is computed as 0
  - netAmount is the same as the totalAmount, even though the fee is configured.

Impact: Users can escape paying fees, causing the protocol to not collect fees when processing withdrawals, and brokers to not earn fees when processing deposits.

Proof of Concept:

```
1  // SPDX-License-Identifier: UNLICENSED
2  pragma solidity >=0.8.22;
3
4  import { UD60x18 } from "@prb/math/src/UD60x18.sol";
5  import { Helpers } from "src/libraries/Helpers.sol";
6
7  import { Test, console2 } from "forge-std/src/Test.sol";
8
9  contract TestEscapePayingFees is Test {
10     UD60x18 public constant FEE = UD60x18.wrap(0.01e18);
11     // 0.1 => 10%
12     // 0.01 => 1%
13
14     function test_calculateAmountsFromFee(
```

```
15          // uint128 totalAmount
16          // UD60x18 fee
17      )
18          external
19          pure
20          returns (uint128 feeAmount, uint128 netAmount)
21      {
22          //@audit => 1 GUSD === 1e2 ===> 100
23              //@audit => 99 units of GUSD ===> ~.99 USD!
24          //@audit-issue => By withdrawing/depositing 99 units of GUSD at
                a time allows users to escape paying fees
25          uint128 totalAmount = 99; //99 units of GUSD
26
27          (feeAmount, netAmount) = Helpers.calculateAmountsFromFee(
               totalAmount, FEE);
28
29          console2.log("feeAmount: ", feeAmount);
30          console2.log("netAmount: ", netAmount);
31
32          assertEq(feeAmount,0);
33          assertEq(netAmount,totalAmount);
34      }
35  }
```

Recommended Mitigation: In the `Helpers.calculateAmountsFromFee()`, check if `fee > 0`, if so, enforce that `feeAmount != 0`, otherwise, revert the tx with an error about `totalAmount` not being above the minimum allowed deposit/withdrawal.

Low

[L-1] Tokens without the decimals() implementation

Description: Under _create() contract is using the `decimals()` method to retrieve erc20 decimals. See erc20 standard `decimals` is not mandatory in the erc20 standard, it was later introduced an an optional feature . So all tokens may not implement it. And for tokens that does not implement it , they can't use the protocol to create streams. It is crucial to note that some tokens may not implement this function, either because they were created before its introduction or simply choose not to include it in order to reduce gas costs or contract complexity. Therefore, attempting to cast all tokens to the IERC20Metadata interface and calling decimals() without proper validation can result in runtime errors and potential vulnerabilities.

```
1  uint8 tokenDecimals = IERC20Metadata(address(token)).decimals();
```

Impact: DoS of stream creation for tokens that don't implement the optional `decimals()` method.

Recommended Mitigation: To avoid this issue and ensure compatibility with a wider range of tokens, it is recommended to follow a more resilient approach. Instead of blindly relying on the decimals() function, consider using other strategies such as:

- Allowing users to specify the number of decimal places explicitly when interacting with token balances or performing calculations.
- Utilizing libraries or utility functions that perform validation and fallback to a default value or heuristic method when decimals() is not available.
- Providing clear documentation and warnings to users on potential compatibility issues when interacting with tokens that may not implement decimals().