Sablier Flow

Security Review

Cantina Managed review by:
Zach Obront, Lead Security Researcher
RustyRabbit, Security Researcher

October 20, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Sablier is a token streaming protocol available on Ethereum, Optimism, Arbitrum, Polygon, Ronin, Avalanche, and BSC. It's the first of its kind to have ever been built in crypto, tracing its origins back to 2019. Similar to how you can stream a movie on Netflix or a song on Spotify, so you can stream tokens by the second on Sablier.

From Oct 7th to Oct 15th the Cantina team conducted a review of flow on commit hash fbf6ff59.

The Cantina team reviewed Sablier's flow changes holistically on commit hash 5dc175cc and determined that all issues were resolved and no new issues were identified.

The team identified a total of **12** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 3
- Low Risk: 7
- Gas Optimizations: 0
- Informational: 1

3 Findings

3.1 High Risk

3.1.1 Sender can brick stream by forcing overflow in debt calculation

Severity: High Risk

Context: SablierFlow.sol#L474

Description: The `_ongoingDebtOf()` internal function is used to calculate the amount of funds owed to the stream recipient since the last snapshot. As a part of these calculations, the `scaledOngoingDebt` is calculated by multiplying the seconds that have passed since the last snapshot by the rate per second.

```
uint128 scaledOngoingDebt = elapsedTime * ratePerSecond;
```

Since `elapsedTime` and `scaledOngoingDebt` are both `uint128`, any result of the multiplication that is greater than `uint128` will overflow and cause a revert.

Note that this multiplication does not require an unrealistically high balance of the token, only for `ratePerSecond` to be set to a high value, which is completely in the control of the `sender`.

This is a major concern because, once this calculation overflows, any calls to `withdraw()`, `refund()`, or to adjust the rate back down will all fail, because they all rely on this function. As a result, once this change happens, there is nothing anyone can do to receive funds from the stream, and all funds will permanently be stuck.

This fact could lead to problems in two situations:

1. It could be abused by a sender who is angry with a recipient to lock all previously streamed funds that have not yet been withdrawn, which should be the property of the recipient.
2. It could occur because a `ratePerSecond` is set to too high of a value accidentally, and then cannot be recovered by either party.

Proof of Concept: The following proof of concept (which can be placed in any file that imports `Integration_Test`) demonstrates the issue:

```
function test_HighRPSRevert() public {
    deal(address(usdc), address(this), DEPOSIT_AMOUNT_6D);
    usdc.approve(address(flow), DEPOSIT_AMOUNT_6D);

    address receiver = makeAddr("receiver");

    uint streamId = flow.createAndDeposit({
        sender: address(this),
        recipient: receiver,
        ratePerSecond: UD21x18.wrap(type(uint128).max),
        token: usdc,
        transferable: true,
        amount: DEPOSIT_AMOUNT_6D
    });

    vm.warp(block.timestamp + 12);

    vm.expectRevert();
    flow.totalDebtOf(streamId);

    vm.expectRevert();
    flow.pause(streamId);

    vm.expectRevert();
    vm.prank(receiver);
    flow.withdraw(streamId, receiver, 1);
}
```

Recommendation: Use a `uint256` for `scaledOngoingDebt`, and carry this type through all functions until the value is compared to the balance. At that point, you can safely downcast to `uint128`.

Sablier: Fixed in PR 296.

Cantina Managed: Confirmed.

3.2 Medium Risk

3.2.1 `isTransferable()` will succeed and return `false` for non-existent streams

Severity: Medium Risk

Context: SablierFlowBase.sol#L184-L186

Description: All the public view functions in `SablierFlowBase.sol` use the `notNull` modifier to ensure that they revert when called for a non-existent stream. For example:

```
function isPaused(uint256 streamId) external view override notNull(streamId) returns (bool result) {
    result = _streams[streamId].ratePerSecond.unwrap() == 0;
}
```

However, the `isTransferable()` function is missing this modifier, so will return `false` instead:

```
function isTransferable(uint256 streamId) external view override returns (bool result) {
    result = _streams[streamId].isTransferable;
}
```

This contradicts the natspec, which says:

```
/// @dev Reverts if `streamId` references a null stream.
```

Proof of Concept:

```
function test_isTransferableDoesNotRevert() public {
    assertEq(flow.isTransferable(2387345), false);
}
```

Recommendation:

```
- function isTransferable(uint256 streamId) external view override returns (bool result) {
+ function isTransferable(uint256 streamId) external view override notNull(streamId) returns (bool result) {
      result = _streams[streamId].isTransferable;
  }
```

Sablier: The fix for this issue was included in PR 296.

Cantina Managed: Confirmed.

3.2.2 New protocol fee applies retroactively on debt before fee change

Severity: Medium Risk

Context: SablierFlow.sol#L811-L820

Description: The `_withdraw()` function applies the protocol fee on the withdrawn amount regardless of whether the debt was accumulated before or after the fee change.

As such, users may be reluctant to use the protocol since:

- The admin can potentially steal 10% from all users at any time
- Even if the admin is not malicious, users can not have confidence in the fee that will be applied up front.

Proof of Concept: Add the following to `withdraw.t.sol`

```
function test_withdrawAfterFeeUpdate() external {
    uint256 fee = flow.protocolFee(usdc).unwrap();
    assertEq(fee, 0);
    uint256 streamId = createDefaultStream(usdc);
    resetPrank({ msgSender: users.sender });
    deposit(streamId, DEPOSIT_AMOUNT_6D);
    vm.warp({ newTimestamp: flow.depletionTimeOf(defaultStreamId)});
    resetPrank({ msgSender: users.admin });
    flow.setProtocolFee(usdc, PROTOCOL_FEE);
    fee = flow.protocolFee(usdc).unwrap();
    assertEq(fee, PROTOCOL_FEE);
    expectCallToTransfer({ token: usdc, to: users.recipient, amount: WITHDRAW_AMOUNT_6D -
↪  PROTOCOL_FEE_AMOUNT_6D });
    flow.withdraw({ streamId: streamId, to: users.recipient, amount: WITHDRAW_AMOUNT_6D });
}
```

Recommendation: Consider storing the fee in effect as part of the stream parameters when created.

Sablier: We appreciate your input. While we agree with your valid points, this is a strategic business decision to facilitate monetization via all pending withdrawals including streamed tokens pre-activation of the protocol fee. Therefore, we have decided to keep the current version.

Cantina Managed: Acknowledged.

3.2.3 Sender can steal deposits from depositors/recipient via refund

Severity: Medium Risk

Context: SablierFlow.sol#L490-L492

Description: Anyone can create a flow and anyone can deposit in to the flow which is used in the case of grants where multiple contributors fund the project as a `recipient`. The sender is specified by the creator of the stream and can be the creator, first depositor or any other actor.

A sender can also `refund` the balance of a stream that exceeds the debt accumulated. As such the funds from other depositors can be stolen by the `sender` if they exceed the current `totalDebt`.

Proof of Concept: Add the following to `refund.t.sol`:

```
    function test_SenderRefundsFromDepositors() external{
    vm.stopPrank();
    deal({ token: address(usdc), to: users.sender, give: 0}); //set sender balance to 0 for ease
    uint128 senderInitialBalance = uint128(usdc.balanceOf(users.sender));
    uint256 streamId = createDefaultStream(IERC20(address(usdc)));
    vm.prank(users.eve);    //as depositor
    flow.deposit(streamId, DEPOSIT_AMOUNT_6D);
    expectCallToTransfer({ token: usdc, to: users.sender, amount: DEPOSIT_AMOUNT_6D });
    vm.prank(users.sender);
    flow.refund({ streamId: streamId, amount: DEPOSIT_AMOUNT_6D });
    uint128 senderBalance = uint128(usdc.balanceOf(users.sender));
    assertEq(senderBalance, senderInitialBalance + DEPOSIT_AMOUNT_6D);
    vm.warp({ newTimestamp: WARP_ONE_MONTH });
    //Withdraw to recipient fails as the balance is 0
    vm.expectRevert(abi.encodeWithSelector(Errors.SablierFlow_Overdraw.selector, streamId, WITHDRAW_AMOUNT_6D,
↪  0));
    flow.withdraw({ streamId: streamId, to: users.recipient, amount: WITHDRAW_AMOUNT_6D });
}
```

Recommendation: Consider adding the functionality to make a flow non-refundable so depositors are confident their contribution will not be taken by the sender.

Sablier: We operate under the assumption that users depositing into any stream trust the stream's owner. Given that Flow streams don't require upfront deposits, there is also an assumption that trust has been established between stream sender, depositors and the recipient. Therefore, depositors are not expected to fund arbitrary streams. Nonetheless, we appreciate your suggestion and may consider offering non-refundable streams in future versions.

Cantina Managed: Acknowledged.

3.3 Low Risk

3.3.1 `depletionTimeOf()` can return incorrect time due to unchecked overflow

Severity: Low Risk

Context: SablierFlow.sol#L86-L95

Description: `depletionTimeOf()` calculates the time that a stream will run out of funds by converting the excess balance into 18 decimals, and then dividing by the rate per second to find the number of remaining seconds. This is performed in an unchecked block.

The first step (converting the excess funds into 18 decimals) can involve multiplying the excess balance by a number as high as `10 ** 18`. In an extreme case (approx 3.4e20 excess tokens with default RATE_PER_SECOND), this value can be high enough to silently overflow.

This results in a very small 18 decimal balance, which results in an even smaller number of remaining seconds after division.

Proof of Concept: The following test can be added to `depletionTimeOf.t.sol` to demonstrate the issue:

```
function test_depletionTimeOfOverflow() public {
    vm.stopPrank();

    IERC20 token = createToken(0);
    uint256 streamId = createDefaultStream(RATE_PER_SECOND, token);

    uint overflowAmt = 340282366920938463464;
    deal(address(token), address(this), overflowAmt);
    token.approve(address(flow), overflowAmt);

    flow.deposit(streamId, uint128(overflowAmt));
    assert(flow.depletionTimeOf(streamId) < block.timestamp + 1 hours);
}
```

Recommendation: `solvencyAmount` should be a `uint256`.

Sablier: We have addressed this issue in PR 296.

Cantina Managed: Confirmed.

3.3.2 `depletionTimeOf()` can fail due to overflowing `uint40` result

Severity: Low Risk

Context: SablierFlow.sol#L94

Description: `depletionTimeOf()` returns a `uint40`. This is common because it measures time, and all reasonable times fit comfortably within that value.

However, `depletionTimeOf()` is calculated by taking the excess balance of an asset, dividing it by the rate per second, and adding this number of seconds to the previous snapshot time. For a very slow stream, there is nothing to prevent a large amount of assets being deposited that would last an amount of time that exceeds the max `uint40`.

In these cases, the `depletionTimeOf()` would revert.

Proof of Concept: The following test can be added to `depletionTimeOf.t.sol`, which demonstrates the effect of depositing just 1 USDC against an extremely slow stream, which causes the function to overflow:

```
function testZach_depletionTimeOfRevert() public {
    vm.stopPrank();

    uint256 streamId = createDefaultStream(UD21x18.wrap(1), usdc);

    uint oneDollar = 1e6;
    deal(address(usdc), address(this), oneDollar);
    usdc.approve(address(flow), oneDollar);

    flow.deposit(streamId, uint128(oneDollar));

    vm.expectRevert();
    flow.depletionTimeOf(streamId);
}
```

Recommendation: `depletionTimeOf()` should return a `uint256`, since the value isn't used internally and therefore doesn't need to match with the type for `snapshotTime`.

Sablier: We have addressed this issue in PR 296.

Cantina Managed: Confirmed.

3.3.3 Reorg attack can steal deposits made shortly after flow creation

Severity: Low Risk

Context: SablierFlow.sol#L240-L243

Description: The `deposit()` function only requires the `streamId` to deposit funds into the flow. It does not provide any safety check on the other parameters of the flow like `sender`, `recipient` or `token`. If an attacker manages a reorg attack they can create their own flow with the initial `streamId`of the victim. Any deposits made based on the initial `streamId` are then directed to the attackers flow.

Note that this also means that if the victim has set an approval for another token with a higher value the attacker can choose that one for their flow and the amount of those tokens will be transferred rather than the original intended ones. This can for instance happen if the depositor already has created other flows with different tokens.

Recommendation: Consider letting the depositor specify the important flow parameters (e.g. `sender`, `recipient` `token` and `transferable`) either in full or as a bundled hash and perform a check.

Sablier: We've decided to address this finding as it could compromise the protocol's security. This issue is resolved in PR 313.

You can see that `deposit` now requires both `sender` and `recipient` to be specified. So if there is a reorg, an attacker cannot steal funds from `deposit` by becoming the stream's beneficiaries. Although you suggested including the `token` as well, we believe doing so would negatively impact user experience. With the new changes, the only risk that remains is that an attacker could potentially change the token address or other stream parameters such as `rps`, but these would still involve the correct sender and recipient. Thus, there is no incentive for the attacker unless he is the recipient, which we consider highly improbable in practice.

Cantina Managed: Confirmed.

3.3.4 `depletionTimeOf()` returns 0 when still solvent at edge of depletion time

Severity: Low Risk

Context: SablierFlow.sol#L75

Description: The Natspec of the `depletionTimeOf()` states that it returns `0` when there is uncovered debt.

```
/// @notice Returns the time at which the stream will deplete its balance and start to accumulate uncovered
↪  debt. If
/// there already is uncovered debt, it returns zero.
```

The `solvencyPeriod` is also calculated based on when the debt exceeds the balance by 1.

```
if (tokenDecimals == 18) {
    solvencyAmount = (balance - snapshotDebt + 1);
} else {
    uint128 scaleFactor = (10 ** (18 - tokenDecimals)).toUint128();
    solvencyAmount = (balance - snapshotDebt + 1) * scaleFactor;
}
uint256 solvencyPeriod = solvencyAmount / _streams[streamId].ratePerSecond.unwrap();
```

Therfore `depletionTimeOf()` should not return `0` when the `totalDebt == balance`, but rather the timestamp at which 1 more token will be streamed.

Recommendation: Change the `depleteiontimeOf()`as follows:

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

Sablier: There was indeed the case where the NatSpec and the actual implementation were not in sync. The fix for this issue was included in PR 296.

Cantina Managed: Confirmed.

3.3.5 Rounding of `snapshotDebt` will cause all streams to be slightly underpaid

Severity: Low Risk

Context: SablierFlow.sol#L483-L485, DataTypes.sol#L74-L75

Description: Currently, the protocol uses 18 decimals when storing the `ratePerSecond` to increase precision when the underlying token has fewer than 18 decimals. This improves the precision of calculation between 2 withdrawals (or any function that alters the `snapshotDebt`).

Whenever `withdraw()` is called, the calculation happens with the increased precision, but then when storing the remaining `snapshotDebt`, we cast back to the token's decimals. The result is that we are increasing the `snapshotTime` to the present, but increasing the debt slightly less than we should be.

Since there is no "final amount" to be streamed, this lowering of the debt is permanently lost to the `recipient` in favor of the `sender`.

How big is this loss? In most cases, it's extremely small. It depends on the exact rounding of the calculation, but will be approximately evenly distributed between 0 and 1 unit of the token (in its own decimals).

- In the case of WETH ($2000, 18 decimals), the max rounding represents `2e-15`.
- In the case of USDC ($1, 6 decimals), the max rounding represents `1e-6`.
- In the case of WBTC ($60000, 8 decimals), the max rounding represents `6e-4`.
- In the case of GUSD ($1, 2 decimals), the max rounding represents `1e-2`.

Each of these values is very small, with the most extreme (GUSD) losing only $0.01 per call to `withdraw()`.

However, with higher value, lower decimal tokens, there is no bound on how extreme this rounding could be. Especially given that `withdraw()` is permissionless, a token with more substantial rounding could provide an incentive for an attacker to call `withdraw()` repeatedly to reduce the recipient's funds.

Recommendation: Store the `snapshotDebt` as a `uint256` in the same full 18 decimal precision as the `ratePerSecond`.

This will ensure that all funds that should go to the recipient are allocated to the recipient.

Then, it is much simpler to ensure that when the recipient claims the funds, the conversion is performed, and their `snapshotDebt` only decreases by an amount that represents the actual funds they claimed.

Sablier: We have implemented the fix to store `snapshotDebt` as a 18-decimal fixed-point number in PR 312. This value is only descaled during withdrawal, thereby improving the system's precision (the snapshot debt as `uint256` was already implemented in PR 296).

3.3.6 `depletionTimeOf()` should not be trusted by on chain integrators

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: `depletionTimeOf()` is a view function that returns the amount of time left before a stream runs out of funds.

Because the sender has complete control over the stream, within a single block they are able to sandwich any call to `depletionTimeOf()` to:

- Make this time far in the future, but temporarily lowering RPS.
- Make this time immediate, by temporarily raising RPS.
- Make this call revert, by pausing the stream.

Recommendation: It should explicitly be made clear to integrating protocols that they should not rely on this function, and its only purpose is to support off chain frontends without any risk to being manipulated.

Sablier: We acknowledge this issue, and we will make it very clear in our docs that this function cannot be fully trusted for on-chain integration. However, it is a useful function for frontend integrations, and we recommend its use in those cases.

Cantina Managed: Acknowledged.

3.3.7 `setProtocolFee()` does not emit `BatchMetadataUpdate` to indicate metadata update

Severity: Low Risk

Context: SablierFlowBase.sol#L262

Description: The current version of the `FlowNFTDescriptor` contract responsible for the metadata JSON does not include the protocol fee as part of the metadata. However a future version of the NFTDescriptor can include the protocol fee and therefore a change of the fee should emit a `BatchMetadataUpdate` event so third parties can update the metadata accordingly. EIP-4906 also specifies that it MUST be emitted in this case:

> The MetadataUpdate or BatchMetadataUpdate event MUST be emitted when the JSON metadata of a token, or a consecutive range of tokens, is changed.

Recommendation: Consider adding the following to `setProtocolFee()`

```
+ emit BatchMetadataUpdate({ _fromTokenId: 1, _toTokenId: nextStreamId - 1 });
```

Sablier: We have not used `BatchMetadataUpdate` because the current descriptor only returns a simple logo, but as you mentioned, updating it in the future could cause issues. It has been addressed in PR 306.

Cantina Managed: Confirmed.

3.4 Informational

3.4.1 `withdrawMax()` does not return the actual received amount

Severity: Informational

Context: SablierFlow.sol#L410-L413

Description: `withdrawMax()` is provided as a convenience function to be called without having to know the `withdrawableAmount` and the `protocolFee` that will be applied if any.

Any integrating contract will most likely need to know the exact amount transferred to the specified `to` address. Therefore to determine the amount (not including the fees) would require before and after balance checks.

Recommendation: Consider returning the amount not including the fees sent to the to address for `withdrawMax` and possibly also for `withdraw` as it too does not give any indication of the fee applied.

Sablier: This issue has been fixed in PR 304. We decided to return both the net amount withdrawn and the protocol fee in the withdrawMax and withdraw functions.

Cantina Managed: Confirmed.