Sablier v2 Protocol Security Review

A security review of the Sablier v2 smart contract protocol was done by Rahul Saxena from **Bluethroat Labs**.
This audit report includes all the vulnerabilities, issues and code improvements found during the security review.

Disclaimer

"Audits are a time, resource and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**."

- Secureum

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behaviour with some of the protocol's functionalities that's not so critical.

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions and the cost of the attack is relatively low to the amount of funds that can be stolen or lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - has too many or too unlikely assumptions or requires a huge stake by the attacker with little or no incentive.

Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

Executive summary

Overview

Project Name	Sablier Finance v2
Repository	https://github.com/sablierhq/v2-core/
Commit hash	8bd57ebb31fddf6ef262477e5a378027db8b85d8

Documentation	Provided
Methods	Manual review

Issues found

Severity	Count
High risk	2
Medium risk	3
Low risk	3
Informational	14
Gas	3
Testing Infra Recommendations	6

Findings

High Severity

[H-1] Assets can remain stuck forever inside the protocol

Context

Presently, there is only one method for the protocol (admin) to claim money from the protocol. For this, they use the `SablierV2Config.claimProtocolRevenues` function which transfers `_protocolRevenues[asset]` amount of ERC20 `asset` to the protocol admin.

This is the function to claim money from the protocol:

```
    function claimProtocolRevenues(IERC20 asset) external override
onlyAdmin {
        // Checks: the protocol revenues are not zero.
        uint128 revenues = protocolRevenues[asset];
        if (revenues == 0) {
            revert Errors.SablierV2Base_NoProtocolRevenues(asset);
        }

        // Effects: set the protocol revenues to zero.
        protocolRevenues[asset] = 0;

        // Interactions: perform the ERC-20 transfer to pay the protocol
revenues.
        asset.safeTransfer({ to: msg.sender, value: revenues });

        // Log the claim of the protocol revenues.
```

```
        emit ISablierV2Base.ClaimProtocolRevenues({ admin: msg.sender,
asset: asset, protocolRevenues: revenues });
    }
```

As we can see, this function can be used to only claim the amount in `protocolRevenues[asset]` and nothing else.

The mapping of `protocolRevenues[asset]` too is only updated at 3 places, inside of `LockupLinear` and `LockupDynamic` while stream creation and inside of the `SablierV2FlashLoan.flashLoan` function.

Description

As seen in the context, the admin can pull out all the fee that the protocol generates and it is quite alright.

However, consider the following cases:

1. Someone randomly or mistakenly sent some ERC20 tokens into the protocol.
2. A stream receiever was deceived into sending their stream NFT to the Sablier contract itself
3. And any other case in which the protocol gets more ERC20 tokens than it should have recieved.

In all the above cases, these funds would be available for a flash loan, assuming they are `flashLoanable`. However these funds can never be taken out of the Sablier contracts.

This would lead to potential loss of assets from the counter-party and Sablier could become liable for legal action.

> **Note:** Even if you try to cancel the stream incase the stream NFT has been transferred to the protocol contracts itself, the ERC20 tokens (the amount that is withdrawable by the recepient) would be sent to the protocol itself and thus become locked forever.

Recommended Mitigation Steps

Set up a new function called `collectDust (IERC20 asset, uint256 amount)` that can be used to pull any extra amount of money apart from the fee out of the contract.

A sample implementation could look like this:

```
function collectDust(IERC20 asset, uint256 amount) external onlyAdmin {
    uint256 assetDust = asset.balanceOf(address(this)) -
protocolRevenues[asset];

    if(assetDust < amount) {
        revert Errors.SablierV2Base_DustLessThanExpected(asset);
    }

    asset.safeTransfer({ to: msg.sender, value: amount });

    // Log the claim of the protocol revenues.
    emit ISablierV2Base.ClaimAssetDust({ admin: msg.sender, asset: asset,
```

```
assetDust: assetDust });
}
```

Extra mitigation step:

Set up a function to make sure that the protocol admin also has the ability to pull out the native tokens of the chain outside from the contract such as ETH, ARB, MATIC, etc.

These native token can be sent into the contracts currently even against the will of the contract. Take a look at force feeding ether

[H-2] Withdraw and Cancel functions would not work for certain tokens

Context

Inside of the contract, `SablierV2LockupLinear` and `SablierV2LockupDynamic`, for the functions `_cancel` and `_withdraw` we see that the `safeTransfer` function has been used for the transfer of funds from the contract to the respective users. For example, see line 26:

```
function _withdraw(uint256 streamId, address to, uint128 amount) internal
override {
        if (amount == 0) {
            revert Errors.SablierV2Lockup_WithdrawAmountZero(streamId);
        }

        uint128 withdrawableAmount = withdrawableAmountOf(streamId);
        if (amount > withdrawableAmount) {
            revert
Errors.SablierV2Lockup_WithdrawAmountGreaterThanWithdrawableAmount(
                streamId, amount, withdrawableAmount
            );
        }

        unchecked {
            _streams[streamId].amounts.withdrawn += amount;
        }

        LockupDynamic.Stream memory stream = _streams[streamId];
        address recipient = _ownerOf(streamId);

        assert(stream.amounts.deposit >= stream.amounts.withdrawn);

        if (stream.amounts.deposit == stream.amounts.withdrawn) {
            _streams[streamId].status = Lockup.Status.DEPLETED;
        }

        stream.asset.safeTransfer({ to: to, value: amount });

        if (msg.sender != recipient && recipient.code.length > 0) {
            try ISablierV2LockupRecipient(recipient).onStreamWithdrawn({
```

```
                streamId: streamId,
                caller: msg.sender,
                to: to,
                amount: amount
            }) { } catch { }
        }

        emit ISablierV2Lockup.WithdrawFromLockupStream(streamId, to,
amount);
    }
```

Description

The interesting thing about `safeTransfer` or (`safeTransferFrom`) is that it does not return a boolean value to represent whether the transfer was successful or not, instead it simply throws when the transfer was not successful.

Combine this fact with the possibility that the protocol does not have the required number of tokens at that point of time or there is an ERC20 token, that puts the restriction inside their `_beforeTokenTransfer` hook that `address(from) != address(to)` or the `amountToBeTransfered != 0` or the particular IERC20 asset in question are **pausable tokens** such as **BNB**, **ZIL**, etc. Then, we have a very real scenario where the sender or recepient cannot withdraw their funds or cancel their streams simply because the `safeTransfer` will revert everytime resulting in the entire `_withdraw` or `_cancel` function reverting.

Recommended Mitigation Steps

The recommended mitigation steps are 2-fold for this particular vulnerability.

1. Consider setting up a whitelist of allowed ERC20 tokens. For a list of what can go wrong with allowing all ERC20 tokens, have a look at this list.
2. For the function `_withdraw`, `_cancel` and for stream-creating functions which have the possibility of reverting because of failed token transfer, use try-catch block and in case the `transfer` reverts, maintain a list of how much money is owed to whom, but execute the rest of the function logic.
3. Additionally, to avoid having to deal with ERC-777 tokens, consider use of the following:

```
error NoERC777();

modifier notERC777(address token) {
    if(
        erc1820Registry.getInterfaceImplementer(
            token,
            keccak256("ERC777Token")
        ) != address(0)
    ) revert NoERC777();
    _;
}
```

Medium severity

[M-1] Malicious tokens can get whitelisted

Context

Consider the following `toggleFlashAsset` function inside of the `SablierV2Comptroller` contract.

```
    function toggleFlashAsset(IERC20 asset) external override onlyAdmin {
        // Effects: enable the ERC-20 asset for flash loaning.
        bool oldFlag = _flashAssets[asset];
        _flashAssets[asset] = !oldFlag;

        // Log the change of the flash asset flag.
        emit ISablierV2Comptroller.ToggleFlashAsset({ admin: msg.sender,
asset: asset, newFlag: !oldFlag });
    }
```

This function is callable only by the admin and is used to toggle the whitelisting or the ability of the `asset` token to be flash-loaned.

Description

As seen in the **Context**, the function `SablierV2Comptroller.toggleFlashAsset` is used to toggle the ability of the `asset` token to be flash-loaned.

However, consider the scenario where *the admin passes a wrong address by mistake, or is tricked into passing a wrong address*.

In that case, an unwanted and most probably a malicious token gets permission to be flash-loaned. Since, the `asset` token is most likely to be malicious, it could return arbitarily high amounts that could be flash-loaned (among other things), for example, see the function below to understand how that is possible:

```
    function maxFlashLoan(address asset) external view override returns
(uint256 amount) {
        // The default value is zero, so it doesn't have to be explicitly
set.
        if (comptroller.flashAssets(IERC20(asset))) {
            amount = IERC20(asset).balanceOf(address(this));
        }
    }
```

> **Note:** This vulnerability has been categorised as a medium severity vulnerability simply because of the fact that the `admin` controls the `SablierV2Comptroller.toggleFlashAsset` function. Otherwise, it would have been a High Severity issue.

Recommended Mitigation Steps

The recommended mitigation steps here are 2 folds:

1. Create a list of whitelisted ERC20 tokens that the protocol allows to use with it and compare all entries of `IERC20 assets` against that list. This checks the *validity* of the passed `IERC20` token.
2. Create two functions: a. Function `toggleFlashAsset` to change the *flash-loanability* of a token b. Function `addNewToken` to introduce a new token in the protocol that can be flash-loaned.

[M-2] Function `withdrawMultiple` can eat up a **LOT** of gas

Context

Consider the following function `SablierV2Lockup.withdrawMultiple`:

```
function withdrawMultiple( uint256[] calldata streamIds, address to,
uint128[] calldata amounts) external override noDelegateCall {
        if (to == address(0)) {
            revert Errors.SablierV2Lockup_WithdrawToZeroAddress();
        }

        uint256 streamIdsCount = streamIds.length;
        uint256 amountsCount = amounts.length;
        if (streamIdsCount != amountsCount) {
            revert
Errors.SablierV2Lockup_WithdrawArrayCountsNotEqual(streamIdsCount,
amountsCount);
        }

        uint256 streamId;
        for (uint256 i = 0; i < streamIdsCount;) {
            streamId = streamIds[i];

            if (getStatus(streamId) == Lockup.Status.ACTIVE) {
                if (!_isApprovedOrOwner(streamId, msg.sender)) {
                    revert Errors.SablierV2Lockup_Unauthorized(streamId,
msg.sender);
                }

                _withdraw(streamId, to, amounts[i]);
            }

            unchecked {
                i += 1;
            }
        }
    }
```

This function can be used by a `recepient` of multiple Sablier streams to withdraw their money from multiple streams all at once and in normal circumstances this would work just as expected.

Description

As explained above, function `withdrawMultiple` would work as expected under normal circumstances. However, consider the following scenario:

1. Alice sends an array of 100_000_000 streamIds along with an array of amounts with same number of entries
2. All of them are valid streamIds, in their ACTIVE state and belong to Alice
3. However, the very last entry, ie, entry number 99_999_999 happen to be in the ACTIVE state but does not belong to Alice
4. In that case, line number 18 from the code snippet would be evoked, ie, `revert Errors.SablierV2Lockup_Unauthorized(streamId, msg.sender);`.
5. This means that the entire transaction will revert and even the 99_999_998 legitimate `_withdraw` actions will **NOT** take place.

This is obviously something which is not very desirable and we should look to mitigate this risk since the arrays `streamIds` and `amounts` are both user-supplied and therefore run a high risk of being malformed or wrong inputs.

Recommended Mitigation Steps

Use try-catch blocks to keep on processing the other streamIds even if some of the streamIds fail the checks. Throw an `event` listing all the streamIds that could not be processed.

[M-3] Single Step Contract Ownership

Context

Consider the following function (`Adminable.transferAdmin`)

```
    function transferAdmin(address newAdmin) public virtual override
onlyAdmin {
        // Effects: update the admin.
        admin = newAdmin;

        // Log the transfer of the admin.
        emit IAdminable.TransferAdmin({ oldAdmin: msg.sender, newAdmin:
newAdmin });
    }
```

Description

The `admin` of the Sablier Protocol can be changed by calling the `transferAdmin` function in `Adminable.sol` contract.

This function immediately sets the contract's new `admin`. Making such a critical change in a single step is error-prone and can lead to irrevocable mistakes.

Recommended Mitigation Steps

Either use the Ownable2Step library from OpenZeppelin or implement your own to mitigate the risks listed in description.

Low severity

[L-1] Redundant Checks

Context

Consider the following functions in `SablierV2Lockup`

Renounce

```
    function renounce(uint256 streamId) external override noDelegateCall
isActiveStream(streamId) {
        // Checks: `msg.sender` is the sender of the stream.
        if (!_isCallerStreamSender(streamId)) {
            revert Errors.SablierV2Lockup_Unauthorized(streamId,
msg.sender);
        }

        // Checks: the stream is not already non-cancelable.
        if (!isCancelable(streamId)) {
            revert
Errors.SablierV2Lockup_RenounceNonCancelableStream(streamId);
        }

        // Effects: renounce the stream.
        _renounce(streamId);
    }
```

Cancel

```
    function cancel(uint256 streamId) external override noDelegateCall
isActiveStream(streamId) {
        // Checks: the stream is cancelable.
        if (!isCancelable(streamId)) {
            revert Errors.SablierV2Lockup_StreamNonCancelable(streamId);
        }

        // Effects and Interactions: cancel the stream.
        _cancel(streamId);
    }
```

Description

In both of the above functions and in any place where the modifier `isActiveStream` and the function `isCancelable` is used in tandem with each other, there is a repeated check of whether the stream with `streamId` is active or not.

That means, `isActiveStream` is checking whether the stream with `streamId` is active or not and also the function `isCancelable` has the following check:

```
    if (_streams[streamId].status != Lockup.Status.ACTIVE) {
        return false;
    }
```

Recommended Mitigation Steps

Limit the usage of the modifier `isActiveStream(streamId)` and then also run your tests again to make sure nothing has broken.

[L-2] No Sanity Check throughout the protocol logic

Context

Consider the following constructors:

SablierV2Comptroller

```
    constructor(address initialAdmin) {
        admin = initialAdmin;
        emit IAdminable.TransferAdmin({ oldAdmin: address(0), newAdmin:
initialAdmin });
    }
```

SablierV2Base

```
    constructor(address initialAdmin, ISablierV2Comptroller
initialComptroller, UD60x18 maxFee) {
        admin = initialAdmin;
        comptroller = initialComptroller;
        MAX_FEE = maxFee;
        emit IAdminable.TransferAdmin({ oldAdmin: address(0), newAdmin:
initialAdmin });
    }
```

Also, consider some functions such as:

SablierV2Comptroller.setFlashFee

```
    function setFlashFee(UD60x18 newFlashFee) external override onlyAdmin
{
        // Effects: set the new flash fee.
        UD60x18 oldFlashFee = flashFee;
        flashFee = newFlashFee;

        // Log the change of the flash fee.
        emit ISablierV2Comptroller.SetFlashFee({ admin: msg.sender,
oldFlashFee: oldFlashFee, newFlashFee: newFlashFee });
    }
```

SablierV2Comptroller.setProtocolFee

```
    function setProtocolFee(IERC20 asset, UD60x18 newProtocolFee) external
override onlyAdmin {
        // Effects: set the new global fee.
        UD60x18 oldProtocolFee = protocolFees[asset];
        protocolFees[asset] = newProtocolFee;

        // Log the change of the protocol fee.
        emit ISablierV2Comptroller.SetProtocolFee({
            admin: msg.sender,
            asset: asset,
            oldProtocolFee: oldProtocolFee,
            newProtocolFee: newProtocolFee
        });
    }
```

Description

In all the constructors and functions described in the **Context** (among others), we see that even basic sanity checks for the user input such as `initialAdmin != address(0)` and `newProtocolFee != 0` are missing.

Granted that it is not very likely that someone passes along a malformed input to these functions or constructors but it is entirely possible if the user is using an unreliable front-end interface.

Although this would not have any devastating consequences, but passing wrong arguments here could result in re-deploying the contracts, leading to economic and time loss for the developers and the users.

> **Note:** Although the sanity checks are missing throughout the protocol contracts and this would usually be a higher severity issue than a low severity one. However, taking other things in context about the protocol and developers, it looks like a concious choice from the protocol developers. In any case, I would still suggest the protocol devs to consider using sanity checks.

Recommended Mitigation Steps

1. Make sure that the users are not able to pass `address(0)` as any input
2. Introduce a concept of minValue/maxValue for things like `setFlashFee`, `setProtocolFee`, etc
3. For values whose upper value is known, such as `MAX_FEE`, make sure that the input is below that value (`1e18` in this case).
4. Consider implementing ERC-165 for inputs such as `comptroller = initialComptroller`

[L-3] `cancel` and `withdrawMultiple` can throw OOG exception

Context

Consider the following example:

```
    function withdrawMultiple(uint256[] calldata streamIds, address to,
uint128[] calldata amounts) external override {
        // Checks: the provided address to withdraw to is not zero.
        if (to == address(0)) {
            revert Errors.SablierV2Lockup_WithdrawToZeroAddress();
        }

        // Checks: count of `streamIds` matches count of `amounts`.
        uint256 streamIdsCount = streamIds.length;
        uint256 amountsCount = amounts.length;
        if (streamIdsCount != amountsCount) {
            revert
Errors.SablierV2Lockup_WithdrawArrayCountsNotEqual(streamIdsCount,
amountsCount);
        }

        // Iterate over the provided array of stream ids and withdraw from
each stream.
        uint256 streamId;
        for (uint256 i = 0; i < streamIdsCount; ) {
            streamId = streamIds[i];

            // If the `streamId` does not point to an active stream,
simply skip it.
            if (getStatus(streamId) == Lockup.Status.ACTIVE) {
                // Checks: `msg.sender` is an approved operator or the
owner of the NFT (also known as the recipient
                // of the stream).

                if (!_isApprovedOrOwner(streamId, msg.sender)) {
                    revert Errors.SablierV2Lockup_Unauthorized(streamId,
msg.sender);
                }

                // Checks, Effects and Interactions: make the withdrawal.
                _withdraw(streamId, to, amounts[i]);
            }

            // Increment the for loop iterator.
            unchecked {
```

```
                i += 1;
            }
        }
    }
```

Description

As seen in the function withdraw in the **Context** heading, the function is used to withdraw from multiple streams in a single function call. However, for a high enough number of entries in `streamIds` and `amounts` arrays, this function will run out of gas and revert with an Out-Of-Gas or OOG exception.

Same is the case with the `SablierV2Lockup.cancelMultiple` function.

Recommended Mitigation Steps

Consider setting up a maximum value of streamIDs that these functions should process in a single go.

Informational

[I-1]

The protocol casts `uint128` to `uint256` and `uint40` to `uint256` among several other castings. While, the logic behind this exact castings is pretty sound and have no reason to overflow, I'd still recommend the developers to consider using OpenZeppelin's SafeCast Library for an added layer of security.

[I-2]

In `DataTypes.sol`, we have the following definition for the possible states of a stream:

```
enum Status {
    NULL,
    ACTIVE,
    CANCELED,
    DEPLETED
}
```

However, consider the time when the `block.timestamp` is past the `endTime` for the stream and the recepient hasn't removed all of their money. The stream would still be in `ACTIVE` state, even though it is past the `endTime`, this could be confusing for end users or developers who build on top of **Sablier Finance**.

[I-3]

In `SablierV2LockupLinear`, we can improve the comment in the `streamedAmountOf` function to include the fact that the cliffTime can also be **equal** to the start time:

```
// If the cliff time is greater than the block timestamp, return zero.
Because the cliff time is always greater than the start time, this also
checks whether the start time is greater than the block timestamp.
```

To, this:

```
// If the cliff time is greater than the block timestamp, return zero.
Because the cliff time is always greater than **or equal to** the start
time, this also checks whether the start time is greater than the block
timestamp.
```

[I-4]

In function `SablierV2LockupLinear._cancel`, instead of calculating the `senderAmount` in this way: `senderAmount = stream.amounts.deposit - stream.amounts.withdrawn - recipientAmount;`, why not simply do `senderAmount = returnableAmountOf(streamId)`?

[I-5]

In my opinion it would be better to inform the users about the inherent risk of approving someone to use their streamNFT because of the following scenario:

1. Suppose A was the intended receiver of a cancelable stream. The NFT was transferred by the approved operator to B.
2. Now, even if we cancel it, the remaining money goes to B, who is not the original intended user.

> The protocol developers could take a step to tackle this by restricting the specific addresses to whom the stream WOULD transfer money.

[I-6]

Consider the function `SablierV2LockupLinear._createWithRange`, here we query the `protocolFee` by calling `comptroller.getProtocolFee(params.asset);`. However, this is assuming that `protocolFee` was actually set in the first place. Maybe you want to know if the answer is not actually 0. Since that could indicate that the fee for this particular asset was never set in the first place.

[I-7]

For the function `SablierV2LockupLinear._withdraw`, the comment *Assert that the withdrawn amount is greater than or equal to the deposit amount.* is wrong and is opposite of what is actually being asserted.

[I-8]

For the function `SablierV2LockupPro.streamedAmountOf`, the comment *If the current time is greater than or equal to the end time, we simply return the deposit minus the withdrawn amount.* is wrong and we should remove the mention of `withdrawn amount` from it. Same for `SablierV2LockupLinear.streamedAmountOf`

[I-9]

The comments above the function `SablierV2LockupPro._calculateStreamedAmountForMultipleSegments` mention the following:

```
/// IMPORTANT: this function must be called only after checking that the
current time is less than the last
/// segment's milestone, lest the loop below encounters an "index out of
bounds" error.
```

This has been mentioned for the following `while` loop:

```
    while (currentSegmentMilestone < currentTime) {
        previousSegmentAmounts += _streams[streamId].segments[index -
1].amount;
        currentSegmentMilestone =
_streams[streamId].segments[index].milestone;
        index += 1;
    }
```

We would recommend the developers to switch to using a bounded `for` loop here if possible. One possible implementation is as follows:

```
    uint index_;
    for(uint i; i < _streams[streamId].segments.length; i++) {
        if(_streams[streamId].segments[index].milestone < currentTime) {
            previousSegmentAmounts +=
_streams[streamId].segments[index].amount;
            index_ = i + 1;
        } else {
            break;
        }
    }
```

[I-10]

Similar to point **I-6**, for function `SablierV2LockupPro._createWithMilestones`, How can you be sure that the fee is 0 or it was never really set?

[I-11]

In all streamNFT creating functions, consider adding a check so that `sender != recipient` and also consider calling *_safeMint* compared to *_mint*.

[I-12]

Same as **I-7** for `SablierV2LockupPro._withdraw`.

[I-13]

In function `SablierV2Lockup.withdraw`, consider the following lines of code:

```
    if (_isCallerStreamSender(streamId) && to != getRecipient(streamId)) {
        revert Errors.SablierV2Lockup_WithdrawSenderUnauthorized(streamId,
msg.sender, to);
    }
```

I recommend the developers to improve the name for this error. Since, the sender will mostly be authorized while the **to** address is not a valid recepient. So, name it something like `SablierV2Lockup_WithdrawRecepientUnauthorized`.

[I-14]

The function `Helpers._checkSegments` is overly restrictive, it is recommended that the following check:

```
    // Check that the deposit amount is equal to the segment amounts sum.
    if (depositAmount != segmentAmountsSum) {
        revert
Errors.SablierV2LockupDynamic_DepositAmountNotEqualToSegmentAmountsSum(
            depositAmount, segmentAmountsSum
        );
    }
```

should be replaced by something less restrictive such as:

```
    if (depositAmount < segmentAmountsSum) {
        revert
Errors.SablierV2LockupDynamic_DepositAmountLessThanSegmentAmountsSum(
            depositAmount, segmentAmountsSum
        );
    }
```

The excess money that you get from the `sender`, can be added to the `returnableAmountOf(streamId)`.

Gas

[G-1]

Declaring constructors as payable can save gas.

[G-2]

Use `external` function modifier instead of `public` wherever possible to save gas. For example: `SablierV2LockupLinear.createWithRange`.

[G-3]

Don't initialize variables to their default value. For example: `uint256 i` is already initialized to 0. No need to re-initialize.

```
for (uint256 i = 0; i < segmentCount; ++i) {
    stream.segments.push(params.segments[i]);
}
```

Testing Infrastructure

After a thorough review of the testing infrastructure of the protocol, it is quite apparent that the protocol developers have put in a **LOT** of thought and care into designing and writing all the tests.

I believe that some of the practices used in Sablier's testing infrastructure have the **potential to become industry standards in testing**, such as the accompanying `.tree` files and the usage of `modifiers` to enforce those tree structures in actual tests.

Here are a few recommendations that I would like to give to the developers of Sablier Protocol.

[TI-1]

Please include the tests wherever it is possible to enforce the heuristic of **source == destination**. For example, I would recommend testing your protocol for the cases, where the sender and the recepient are the same addresses (or recepient transfers their streamNFT to the sender after getting it)

[TI-2]

Test for the cases where the stream recepient transfers their stream NFT to the protocol contract themselves and see if the protocol is equipped to handle this scenario in an equitable manner.

[TI-3]

In most of the test case, DAI is used by default to run all your tests. I would recommend testing your protocol against some tokens with decimals as low as 2 (Gemini USD) and with some tokens with decimals as high as 24 (YAM-V2)

[TI-4]

Test your protocol to see if it is not breaking the ERC 4337 standard. That means, your protocol should work as intended even if the sender or the receiver are using ERC-4337 wallets.

[TI-5]

Please improve the inline code documentation of the tests. The inheritence structure between different test files is quite complex and it would be quite difficult for someone new to the codebase to quickly test their

hypothesis by making a few changes and running your tests.

[TI-6]

This might be hard, but try creating a visualisation (a large tree like structure) of your testing suite, that basically shows all possible scenarios and all possible states the protocol could be in and what are the relevant tests for that particular scenario.

Protocol Logic Review

Part of our audits involves analysis of the protocol and its logic. *Rahul Saxena* from the **Bluethroat Labs** team went through the implementation and documentation of the implemented protocol.

Complete tests and partial documentation had been provided to us, and along with the function and inline documentation, it was sufficient to fully understand the intended protocol that is being implemented.

According to my analysis, the protocol and logic are working as intended, given that the findings listed in the Issues section are fixed. The safety of the protocol is also dependent quite a lot on the safety of the PRBMath Library. This library has **NOT BEEN AUDITED** by the Bluethroat Labs team, but on first impressions, it looks like a solid library with heavy testing to supplement it.

We were not able to discover any additional problems in the protocol implemented in the protocol smart contracts.