Sablier Finance

Security Review

Cantina Managed review by:

Zach Obront, Security Researcher

RustyRabbit, Security Researcher

Taek Lee, Junior Security Researcher

June 8, 2023

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	*Directly* exploitable security vulnerabilities that need to be fixed.
High	Security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All high issues should be addressed.
Medium	Objective in nature but are not security vulnerabilities. Should be addressed unless there is a clear reason not to.
Low	Subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
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. When determining the severity one first needs to determine whether the finding is subjective or objective. All subjective findings are considered of Minor severity.

Next it is determined whether the finding can be regarded as a security vulnerability. Some findings might be objective improvements that need to be fixed, but do not impact the project's security overall (Medium).

Finally, objective findings of security vulnerabilities are classified as either critical or major. Critical findings should be directly vulnerable and have a high likelihood of being exploited. Major findings on the other hand may require specific conditions that need to be met before the vulnerability becomes exploitable.

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 March 20 to March 29 the Cantina team conducted a review of v2-core on commit hash 8bd57e. The team identified a total of **23** issues in the following risk categories:

- Critical Risk: 1
- High Risk: 3
- Medium Risk: 4
- Low Risk: 11
- Gas Optimizations: 2
- Informational: 2

3 Findings

3.1 Critical Risk

3.1.1 Any token with large volume flash minting capability can be stolen

Severity: Critical Risk

Context:

- sd59x18/Math.sol#L596609
- SablierV2LockupDynamic.sol#L240-L249
- SablierV2LockupDynamic.sol#L561-L609

Description: In `SablierV2LockupDynamic.sol`, the amount of a token that is available to be withdrawn is calculated in the following function:

```
function withdrawableAmountOf(uint256 streamId)
    public
    view
    override(ISablierV2Lockup, SablierV2Lockup)
    returns (uint128 withdrawableAmount)
{
    unchecked {
        withdrawableAmount = streamedAmountOf(streamId) - _streams[streamId].amounts.withdrawn;
    }
}
```

This function uses an `unchecked` block, which assumes that `streamedAmountOf(streamId)` will always be less than `_streams[streamId].amounts.withdrawn`.

This is an important invariant of the protocol: streamed amount can never go down, so you can never have withdrawn more than the current streamed amount.

However, due to the issue found in H-01, there is an underlying math bug that allows for the following:

- `x > y`
- `a > 0`
- `x ** a < y ** a`

(See H-01 for more details on this issue, as well as a proof of concept showing example values where this holds.)

The result is that when `streamedAmountOf()` is calculated, it can return a lower multiplier (and therefore a lower resulting value) at a higher timestamp:

```
SD59x18 elapsedSegmentTimePercentage = elapsedSegmentTime.div(totalSegmentTime);
SD59x18 multiplier = elapsedSegmentTimePercentage.pow(currentSegmentExponent);
SD59x18 segmentStreamedAmount = multiplier.mul(currentSegmentAmount);
```

Here is a link to a test that can be dropped into `streamedAmountOf.t.sol` to show an example where, as `block.timestamp` increases by 12 (ie one block later), the resulting `streamedAmountOf()` value decreases: gist link

While many of the `assert` statements protect against the worst possible examples of this, the fact that many of them (and the calculations that lead up to them) are in `unchecked` blocks opens the door for overflows and underflows, which makes an exploit possible.

The precondition for this exploit is that any token allows flash minting of `type(uint128).max` tokens. While this doesn't appear to be the case with any mainstream tokens today, it seems almost a certainty that this will be the case with some established tokens in the future.

With this ability, the following exploit path becomes possible:

1) Create a stream for the given asset with a start time that will lead to the math bug being triggered at a time in the future.
2) The block before the bug is triggered, withdraw the maximum assets from the stream using the `withdrawMax()` function, which will lead to `withdrawn == streamedAmountOf()`.

3) The next block, `streamedAmountOf()` will decrease by 1. As a result, `withdrawableAmountOf()` will underflow (since it's in an unchecked block) and result in a withdrawable amount of `type(uint128).max`.

4) Now the exploit begins. We take a flash mint of just below `type(uint128).max` of the asset and start a new stream, pushing the total amount of the asset in the contract up to `type(uint128).max`. This stream will not be used, it only exists to increase the contract balance sufficiently for the exploit to work.

5) We now call `withdraw()` on our original stream, with an `amount` of `type(uint128).max`. This passes the check that it's within the bounds of our withdrawable amount, but when `withdrawn` is incremented, it overflows, leaving it below `deposit` so it passes the assert check.

```
// Effects: update the withdrawn amount.
unchecked {
    _streams[streamId].amounts.withdrawn += amount;
}

...

// Assert that the withdrawn amount is greater than or equal to the deposit amount.
assert(stream.amounts.deposit >= stream.amounts.withdrawn);
```

6) `type(uint128).max` of the asset is transferred to us, which will pass because we've deposited sufficient assets into the contract that this transfer is possible.

7) We return the flash mint and profit with the full quantity of the asset that previously existed in the contract. All other streams using this asset are now unusable because we've taken all their assets.

Recommendation: Fixing the math bug described in H-01 will resolve this issue, but the exploit is only possible because of the extensive use of `unchecked` blocks.

It is recommended to reduce the use of `unchecked` blocks around crucial invariants, as this can cause assertions to pass without protecting the intended invariants.

3.2 High Risk

3.2.1 Recipient can block the sender's cancel by sending the NFT to an address known to revert the transfer of the underlying ERC20

Severity: High Risk

Context:

- SablierV2LockupDynamic.sol#L420-L433
- SablierV2LockupLinear.sol#L350-L363

Description: The `_cancel()` functions of the `SablierV2LockupDynamic` and `SablierV2LockupLinear` transfer the remaining funds for the sender and recipient in one call using the `safeTransfer` function for each:

```
function _cancel(uint256 streamId) internal override onlySenderOrRecipient(streamId) {
    address recipient = _ownerOf(streamId);
    ...
    if (recipientAmount > 0) {
        ...
        stream.asset.safeTransfer({ to: recipient, value: recipientAmount });
    }
    if (senderAmount > 0) {
        stream.asset.safeTransfer({ to: sender, value: senderAmount });
    }
}
```

`safeTransfer` will revert when the underlying transfer fails in any way. As the recipients address is determined by the ownership of the Sablier NFT, the recipient can front run the sender's cancel transaction by sending the NFT to an address known to revert by the underlying token's `safeTransfer` (e.g. an address on USDC's blocklist).

While this may not directly benefit the recipient, one could easily imagine a situation where a sender decides to cancel a stream, and a recipient is unhappy about it. In this case, they could call `withdraw()` to

withdraw the full amount they are owed, and transfer the NFT to such an address, bricking the sender's funds.

Recommendation: Split the cancel functionality in two separate transactions. One where the initiator stops the stream accounting and withdraws their part of the funds. Then a second transaction where the other party withdraws their funds. This way The transfer to the recipient cannot block the sender's cancel.

Sablier: Fixed in PR 422.

Cantina: Confirmed.

3.2.2 If MAX_FEE is set too high, user may be able to underflow deposit value to steal all funds

Severity: High Risk

Context: SablierV2Comptroller.sol#L45-45

Description: This vulnerability has some narrow constraints that make it unlikely to occur at the moment but would result in theft of all user funds, so are important to address.

There are three constraints required for this attack:

- `MAX_FEE` is set to `1e18` (which is the only value explicitly called out in the comments)
- There is a token that allows flash mints of up to `type(uint128).max` (currently DAI allows 500mme18, but this possibility seems likely at some point)
- This token has a protocol fee that is set extremely low

If those constraints are met, we are able to use them to create a stream with parameters that will underflow deposits and allow them to steal all funds.

Here is a walkthrough of the attack:

1) Take a flash mint of type(uint128).max This is a precondition of the attack, which does not seem to be possible with any mainstream tokens at the moment.

However, there is no reason to assume this will continue to be the case.

2) Create a new stream with this full amount We create a new stream with `type(uint128).max` as the `totalAmount`, our own address as the broker, broker fee set to `1e18 - protocolFee + 1`, and a start time in the past.

(Note that the `totalAmount` value can be adjusted slightly to ensure less liquidity is wasted to `protocolFees`, but we will use these simple values for this proof of concept.)

The function then calls out to `Helpers.checkAndCalculateFees()` to get the exact amounts:

- The function checks that `brokerFee < maxFee`, which will pass if `MAX_FEE = 1e18`.
- We calculate the broker payment amount (which will be slightly below the max uint128).
- Then the following check is performed:

```
unchecked {
    assert(totalAmount > amounts.protocolFee + amounts.brokerFee);

    // Calculate the deposit amount (the amount to stream, net of fees).
    amounts.deposit = totalAmount - amounts.protocolFee - amounts.brokerFee;
}
```

- Because this check is performed in an unchecked block, `protocolFee + brokerFee` will overflow and pass the assertion.
- Then `deposit` will be calculated as `total - protocol - brokerFee`, which will underflow and result in a value slightly under the max uint128.

Once these values are returned, assets are transferred:

- `deposit + protocolFee` is transferred into the contract, which is also in an unchecked block and will overflow, leading to a value of `protocolFee - 1`.

- `brokerAmount` is transferred to another account we own.

3) Steal all user funds At this point, we have the following values:

- `protocolFee` is in the contract (small value)
- `brokerFee` is in another account we own
- `deposit` value on the contract is close to `type(uint128).max`

Since our start time was set to the past, some portion of that `deposit` value is available for immediate withdrawal. (We could even set endTime in the past so that the full amount is available.)

Since our deposit value is now extremely high, we can pass the assertions meant to keep the protocol safe, such as:

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

We can simply call the `withdraw()` function with the total amount of the asset owned by the contract, and we will pass all checks to be able to withdraw these assets (which belong to other users).

4) Pay back flash mint As long as `stolenMoney > protocolFee`, we will have more money than we initially deposited. At this point, we can pay back the flash mint and keep the profits.

5) Continue to monitor the contract This attack completes with a large leftover `deposit` balance. If any other user creates a stream using the same asset in the future, it will be trival to withdraw some portion of the `deposit` balance, emptying the contract of the asset and stealing their stream.

Recommendation: We have a few recommendations in how to best address this:

1) Ensure that `MAX_FEE` is always set to a value far below `1e18`. We would recommend a value less than `0.2e18`.
2) This exploit was possible because of assertions being performed in unchecked blocks. While we understand the gas savings this provides, the security trade-off of performing crucial checks on unchecked arithmetic does not seem worthwhile.

Sablier: We implemented both recommendations in PR 399.

Cantina: Excellent! I agree with the comment in the PR that the extra gas cost is certainly worth it. This fix seems very thorough and will definitely protect against any possible risks that the assertions in the stream creation flow don't actually hold.

3.2.3 PRBMath `pow()` function can return inconsistent values

Severity: High Risk

Context: sd59x18/Math.sol#L596-L609

Description: PRBMath's `pow()` function takes in two signed integers, x and y, and returns x ** y.

```
function pow(SD59x18 x, SD59x18 y) pure returns (SD59x18 result) {
    int256 xInt = x.unwrap();
    int256 yInt = y.unwrap();

    if (xInt == 0) {
        result = yInt == 0 ? UNIT : ZERO;
    } else {
        if (yInt == uUNIT) {
            result = x;
        } else {
            result = exp2(mul(log2(x), y));
        }
    }
}
```

A proper implementation of the `pow()` function should uphold the following invariant:

If `x >= y` and `a >= 0`, then `x ** a >= y ** a`.

This is a crucial invariant for the Sablier protocol, because the `pow()` function is used by `SablierV2LockupDynamic.sol` to compute total streamed amounts based on the current time. It is required for the protocol to function as expected that, as `block.timestamp` increases, the total streamed amount increases as well. Implications of this issue are addressed in C-01 of this report.

- **Proof of Concept**

```
function testPow() public {
    uint128 high = 505456470057136353;
    uint128 low = 505456461792312955;
    assert(high > low);

    SD59x18 exp =  UD2x18.wrap(31414735).intoSD59x18();
    SD59x18 highPow = high.intoSD59x18().pow(exp);
    SD59x18 lowPow = low.intoSD59x18().pow(exp);

    assert(SD59x18.unwrap(highPow) < SD59x18.unwrap(lowPow));
}
```

- **Root Cause Analysis**

In the `exp2()` function, bit masks are applied to multiply the result by `root(2, 2^-i)` for each position `i` with a value of 1.

However, at Common.sol#L170, the value is incorrectly included as `0xFF00000000` instead of `0xFF000000`.

This bug was introduced in PR 77 on April 17th, 2022.

Recommendation: Adjust the comparison value in the `exp2()` function to ensure the calculation is accurate.

Sablier: Changed applied in PR 432.

This PR updates PRBMath to v4, which addresses the critical bug discovered in the Common.exp2 function. Additionally, it includes a test to verify the monotonicity of the streamedAmountOf function refer to PR 421.

I am due to fix another rounding error in the mul and div functions of SD59x18.

Although I was unable to address this issue due to time constraints, I no longer classify it as an "error." As noted in the v4 release notes, I have revised the NatSpec comments to clearly specify the rounding modes in PRBMath: UD60x18 rounds down, while SD59x18 rounds toward zero.

Cantina: Confirmed.

3.3 Medium Risk

3.3.1 Withdraw by the sender when the NFT is in a Defi contract may lead to lost funds

Severity: Medium Risk

Context:

- SablierV2Lockup.sol#L182-L191
- SablierV2Lockup.sol#L63-L68

Description: The Sender has the ability to call the `withdraw()` at any time on behalf of the recipient. This is a functionality created to accommodate the scenario where the stream is created with a recipient address that later turns out to be unable to call the withdraw function itself.

```
function withdraw() public ... isAuthorizedForStream(streamId) {
    ...
}

modifier isAuthorizedForStream(uint256 streamId) {
    if (!_isCallerStreamSender(streamId) && !_isApprovedOrOwner(streamId, msg.sender)) {
        revert Errors.SablierV2Lockup_Unauthorized(streamId, msg.sender);
    }
 _;
}
```

However, this also lets the sender force the withdrawal of the ERC20 tokens on the owner's address of the NFT at that time. This can be with the malicious intent of the sender, but also accidentally when the sender has multiple streams to their employees and decides to call withdraw on a regular basis. The NFT can in such cases be held by a (Defi) contract that may not be able to properly account for the unforeseen increase in underlying ERC20 tokens. In the worst case, the tokens would be locked in the contract or they would be distributed evenly over all participants of the contract (pool or vault type contract).

Recommendation: As the main scenario for this feature is to account for the inability of the recipient to call withdraw, consider changing to a design where the recipient is required to accept the stream before the stream actually can begin. For this, a separate `accept()` function can be created or simply let the recipient call the withdraw function with a zero amount which is a more explicit check of the fact the address can later call the same function. If this `accept`function is not called allow the sender to cancel the stream without cost (protocol and broker fee) and create a new stream to an address that is able to call `withdraw()`.

Sablier: This issue falls under the category of unavoidable problems stemming from the universal programmability of Ethereum. It is impossible to prevent the logic of a contract from being misused by other contracts. A good example that comes to mind is the transfer of ERC-20 tokens to contracts that cannot withdraw them.

> held by a (Defi) contract that may not be able to properly account for the unforeseen increase in underlying ERC20 tokens

The onus is on the developer of that contract to implement the `ISablierV2LockupRecipient` interface, and account for any potential withdrawals by implementing the `onStreamWithdrawn` hook. To assist integrators, we will provide a few ready-to-use templates.

> account for the inability of the recipient to call the withdraw,

Might you have meant to say something like "inability of the recipient to account for withdrawals initiated by senders"? The issue is about when the sender calls `withdraw`.

> consider changing to a design where the recipient is required to accept the stream before the stream actually can begin. For this a separate `accept()` function can be created

Interesting idea, but it only partially addresses the issue.

While the initial recipient may accommodate sender-initiated withdrawals, a subsequent NFT transfer could result in a new recipient contract lacking this ability. A comprehensive solution would require overriding the ERC-721 `safeTransferFrom` function and extending its functionality to ensure the new owner complies with both `ERC721TokenReceiver` and `ISablierV2LockupRecipient`. `transferFrom` would remain unaltered to reflect the intended unsafe behavior in ERC-721.

However, we are not big fans of these "safe transfer" functions as they break composability with smart contract wallets that might not explicitly implement specific interfaces, even though they are perfectly capable of handling the underlying assets.

Therefore, we will treat this issue as an inescapable limitation on the capacity of smart contracts to communicate with each other. We will mitigate its impact by raising awareness through our documentation site and user interface.

Cantina: Acknowledged.

3.3.2 Protocol and Broker are overpaid in the event of a canceled stream

Severity: Medium Risk

Context:

- SablierV2LockupDynamic.sol#L516-L527
- SablierV2LockupLinear.sol#L425-L446

Description: When the stream is created, both the broker fee and the protocol fee are immediately paid in full.

This means that when a stream is canceled, both fees have been paid in full by the sender.

This does not seem ideal for any situation in which the stream is canceled early, but there are two situations where it is particularly harmful:

1) If the stream is canceled before the start time (for example, if parameters were set incorrectly), so no actions were taken at all.
2) The Sablier team mentioned potential functionality where a periphery contract would change the parameters of a stream by canceling and restarting a new stream. In this case, the sender would have to pay the protocol fees in full each time a change was made.

Additionally, it is worth considering that the protocol fee is calculated on the total amount of the stream (including the broker fee). Therefore, it seems that front-ends/brokers would be better served by creating a second transaction for their broker fee, generating the same result but at less cost to the sender or more revenue for the broker.

Recommendation: Gradually stream the broker and protocol fee, either by including it in the stream with the same parameters and release schedule, or by using a separate linear stream with the same start time as the stream they are paying for.

Sablier: While we agree that this is an issue, we're willing to live with it because there is no easy way to address it. Creating a "shadow" fee stream alongside each stream would involve a massive refactor (which we don't have time for), and would also significantly increase the gas cost profile of the protocol.

Furthermore, we argue that this is more of a "Severity: Low Risk" issue, because of the following reasons:

1. The protocol fees will be set to zero for a long time, and even when they are set, they will be set to reasonable values.
2. In practice, the restart functionality of the periphery would only be used occasionally. Based on the feedback from Sablier V1 users, editing streams would be only an occasional event, e.g. when the terms of a vesting agreement change, or when an employee gets a bonus.
3. In the worst-case scenario, when users complain about this issue day in and day out (because they keep getting charged multiple times), we can just cut the protocol fees in half. Doing so would effectively mean we make similar revenues while users are charged the same as if cancelations weren't frequent.
4. Generally speaking, it is common for users/ customers to pay for their errors when they make them.

Cantina: Acknowledged.

3.3.3 Reentrancy risk in `_cancel()` for interacting protocols

Severity: Medium Risk

Context: SablierV2LockupLinear.sol#L333-392

Description: The `_cancel()` function is used to end a stream, sending the total streamed amount to the recipient and the remaining funds back to the sender. It also implements the `onStreamCanceled()` callback function for protocols that interact with Sablier.

The basic flow of the function is as follows:

- Calculate the funds to send to each party
- Send recipient funds to the recipient
- Send sender funds back to the sender
- If the function is being called by the sender, notify the recipient with a callback
- Otherwise, notify the sender with a callback

For any tokens that implement callbacks themselves (for example, ERC777s), this creates a reentrancy risk, because control flow is being passed to the sender before the recipient is sent the callback to update state.

As a simple example, let's imagine an interacting protocol that holds Sablier NFTs as collateral in publicly investable vaults:

- A new stream is posted as collateral on this protocol, and valued according to its total maturity value
- The protocol's vault adjusts its internal "assets" calculations to account for this increased value
- When the stream is canceled, the remaining streamed funds are sent to the protocol

- Then the sender receives their funds, but because the token implements a callback in the transfer method, control flow is passed to the sender
- The sender is then able to enter into the protocol at a time when the value of the assets has changed, but the accounting hasn't yet been updated
- As a result, they may be able to liquidate a position, short the vault, or perform any other action that benefits from the protocol's incorrect accounting
- Only after the sender takes this action is the callback called and the protocol's accounting is updated

Recommendation: Sending funds and calling the callbacks should be batched together, rather than alternating. For example:

- Send recipient funds to the recipient
- If the function is being called by the sender, notify the recipient with a callback <= these two switch
- Send sender funds back to the sender <= these two switch
- If the function is not called by the sender, notify sender with a callback

This will make the `if else` statement slightly less convenient, but is a worthwhile change for the added security.

Sablier: PR 422 has implicitly addressed this issue. Because the `cancel` function does not auto-withdraw the funds to the recipient anymore, this step no longer applies:

> The sender is then able to enter into the protocol at a time when the value of the assets has changed, but the accounting hasn't yet been updated

This means that there is no reentrancy risk in the cancel function anymore.

Cantina: Verified.

3.3.4 Protocol fees can be skirted because of default 0 value

Severity: Medium Risk

Context: SablierV2Comptroller.sol#L45-45

Description: Sablier's protocol fees are tracked in the Comptroller contract on a per-asset basis, stored in the following mapping:

```
mapping(IERC20 asset => UD60x18 fee) public override protocolFees;
```

When a new stream is created, the protocol fees are pulled by accessing this mapping through the public getter function:

```
UD60x18 protocolFee = comptroller.protocolFees(params.asset);
```

As a result, any assets without a fee set will default to a value of zero.

Given the BUSL license and `noDelegateCall` checks, it appears there is a strong effort being made to avoid using the protocol's code and logic without following the expected behavior and paying fees.

However, this issue can be abused to stream any asset with no fees by wrapping it in another contract that allows 1-for-1 deposits and withdrawals of the underlying token, and streaming this contract's token instead.

In an extreme case, it would even be possible for someone to spin up a `FreeSablier.sol` contract, which takes in any asset, creates a proxy, starts a stream with the proxy's token, and distributes everything back to the users.

Recommendation: Implement a default fee to use as a backup when a token's fee is not set:

```
function getProtocolFee(address asset) public view returns (UD60x18) {
    UD60x18 _savedFee = protocolFees[asset];
    return _savedFee == 0 ? defaultFee : _savedFee;
}
```

Sablier: This is a good catch, but it's more like a "Severity: Low Risk" type of issue, because we have the ability to upgrade the comptroller via the setComptroller function. We could easily address this issue even after deployment if it proves to be a significant hurdle.

We will closely monitor the usage of these wrapper contracts and adjust the protocol fee to match that of the underlying token for any wrapper that grows to have significant adoption.

Although a FreeSablier.sol contract is technically feasible, in practice we expect the incentives to be in favor of paying the fee, due to the following reasons:

1. The protocol fees are not expected to be excessively high, which means that the gas expenses for interacting with this proxy system may exceed the cost of simply paying the fee in the first place.
2. We can implement measures to deter any such contract from gaining traction, for example, by hiding all streams created via FreeSablier.sol on the Sablier Interface.
3. Implement a default fee to use as a backup when a token's fee is not set

Adopting this suggestion would conflict with our business intention of letting the default protocol fee be zero for all tokens. Additionally, w intend to maintain all fees at zero during the initial stages of the protocol.

Cantina: We believe this is an important risk if there is ever to be a goal for the protocol to make money from fees. However, it is acceptable to us for Sablier to ignore this issue for now and only address it if it later becomes a concern because:

1) It poses no security threat to users, and only to the protocol's fees.
2) The Comptroller is upgradeable, so Sablier can choose to fix this later if and when it becomes an issue.

Because of the upgradeability of the Comptroller, we are adjusting this issue from High Severity to Medium Severity.

3.4 Low Risk

3.4.1 Assert in `_calculateStreamedAmountForMultipleSegments` can make `withdraw` and `cancel` revert locking funds in the contract.

Severity: Low Risk

Context:

- SablierV2LockupDynamic.sol#L342
- SablierV2LockupDynamic.sol#L367
- SablierV2LockupLinear.sol#L229

Description: In the `_calculateStreamedAmountForMultipleSegments()` and `_calculateStreamedAmount-ForOneSegment()` functions of the `SablierV2lockupDynamic` and the `streamedAmountOf()` function of the `SablierV2LockupLinear` contract there are `asserts` statement to ensure that the calculated amount for the segment does not exceed the configured amount for that segment.

```
function _calculateStreamedAmountForMultipleSegments()
{
    ...
    assert(segmentStreamedAmount.lte(currentSegmentAmount));
}
```

Although this should never happen, if it does this would make the `withdraw` and `cancel` functions also revert. The funds in the stream would then be locked in the contract until that segment has passed which could be a very long time (i.e. a year or more). As this is only about the last segment it would be better in such a case to not count this segment and let `segmentStreamedAmount` be 0. This would allow the recipient to withdraw everything up to the previous segment end. At the end of this segment the full amount of that segment shouldn't cause a problem anymore and in case the segment is very long the sender could cancel the stream to retrieve the remaining funds. This may not be ideal for the recipient but it's better than having the funds stuck.

Recommendation: Consider replacing the `assert` with an `if` statement setting the `segmentStreamedAmount` to `0` when it would exceed the `currentSegmentAmount`.

```
- assert(segmentStreamedAmount.lte(currentSegmentAmount));
+ if (segmentStreamedAmount.gt(currentSegmentAmount)) {
+   segmentStreamedAmount = 0;
+ }
```

Sablier: We implemented the recommendation in this PR 420.

Cantina: Confirmed.

3.4.2 `toggleFashAsset()` does not specify the target state.

Severity: Low Risk

Context:

- SablierV2Comptroller.sol#L89

Description: The Comptroller's `toggleFlashAsset(IERC20 asset)` doesn't specify the target state of the toggle. It's best practice to specify the target state as this is more clear and doesn't depend on the current state.

Recommendation: Define the function including the boolean flag as parameter.

```
function toggleFlashAsset(IERC20 asset, bool flag)
```

Sablier: We're happy to keep this function as is, due to a couple of reasons:

1) This is subjective, but we like the UX of letting the protocol switch the flag better than providing an explicit flag. The name of `toggleFlashAsset` would also be less exact with two parameters, because no toggle would actually occur when the flag is re-set.
2) Fewer arguments to encode makes this function slightly easier to use in a multisig context. `toggleFlashAsset(asset)` is more gas efficient than `toggleFlashAsset(asset,flag)`. I checked this with these Solidity contracts.

Cantina: Acknowledged.

3.4.3 Use of signed SD59x18 in `SablierV2LockupDynamic`

Severity: Low Risk

Context:

- SablierV2LockupDynamic.sol#L319-L346
- SablierV2LockupDynamic.sol#L319-L346

Description: The `SablierV2LockupDynamic` contract uses the signed version of the `SD59x18` library in contrast to the `SablierV2LockupLinear` contract which uses the more appropriate unsigned `UD60x18` version.

The `_calculateStreamedAmountForMultipleSegments()` and `_calculateStreamedAmountForOneSegment()` functions represents the `elapsedSegmentTimePercentage`, `multiplier` and `segmentStreamedAmount` as signed decimals and then ultimately casts the signed version into an unsigned `uint128` (which indicates an assumption of the value being positive).

The main reason for using SD59x18 is because the `pow` function uses `log2` underneath which returns a negative number for `elapsedSegmentTimePercentage` values smaller than one (1e18) - which should always be the case.

```
function _calculateStreamedAmountForMultipleSegments(uint256 streamId) {
    SD59x18 currentSegmentAmount = _streams[streamId].segments[index - 1].amount.intoSD59x18();
    SD59x18 currentSegmentExponent = _streams[streamId].segments[index - 1].exponent.intoSD59x18();
    currentSegmentMilestone = _streams[streamId].segments[index - 1].milestone;
    ...
    SD59x18 elapsedSegmentTime = (currentTime - previousMilestone).intoSD59x18();
    SD59x18 totalSegmentTime = (currentSegmentMilestone - previousMilestone).intoSD59x18();
    SD59x18 elapsedSegmentTimePercentage = elapsedSegmentTime.div(totalSegmentTime);
    SD59x18 multiplier = elapsedSegmentTimePercentage.pow(currentSegmentExponent);
    SD59x18 segmentStreamedAmount = multiplier.mul(currentSegmentAmount);

    assert(segmentStreamedAmount.lte(currentSegmentAmount));
    streamedAmount = previousSegmentAmounts + uint128(segmentStreamedAmount.intoUint256());
```

The risk is, however, that there may be edge cases where `elapsedSegmentTime`, `totalSegmentTime` or `currentSegmentExponent` become negative. All of this math occurs in an `unchecked` block, so the subtractions of `uint128`s would not revert on overflow or underflow.

We haven't found any edge cases where this would be possible, but if `segmentStreamedAmount` would become negative it could also pass the assertion to check whether the calculated `segmentStreamedAmount` is less than the `currentSegmentAmount` set in the segment. The cast could then turn the returned value in a larger number than the `currentSegmentAmount`, which has the potential to slip through the unchecked subtraction and addition in the `_cancel()` function and allow users to withdraw beyond the deposited amount.

Another way the returned value of `streamedAmount` could be different than expected is when the exponent is negative in the calculation of the `multiplier` as the `elapsedSegmentTimePercentage < 1e18`. Again, this shouldn't be possible since `currentSegmentExponent` is a cast of the `segment.exponent` which itself is a `UD2x18` (i.e. `uint64`), but this shows another example of where an issue elsewhere in the code could be exacerbated by unnecessary use of signed integers.

Recommendation: Use the UD60x18 for all variables replace the calculation of the multiplier as follows:

```
- SD59x18 multiplier = elapsedSegmentTimePercentage.pow(currentSegmentExponent);
+ UD60x18 multiplier = UNIT.div(UNIT.div(elapsedSegmentTimePercentage).pow(currentSegmentExponent));
```

This assumes the `elapsedSegmentTimePercentage` is smaller than 1e18, which should always be the case, but a check can be added for certainty.

Sablier: We agree that it would be nice to be able to use UD60x18 in SablierV2LockupDynamic. However, despite our best efforts to find a solution, it turns out that this cannot be done without significantly limiting the scope of the dynamic contract.

https://github.com/PaulRBerg/prb-math/pull/182 to apply your recommendation when the base input is less than 1e18. The modified implementation performs well within the scope of PRBMath, but not when applied to Sablier, because of the following mathematical constraint:

- The base, elapsedTimePercentage, is frequently a very small value.
- Consequently, the variable i becomes exceedingly large.
- As a result, the mul operation that computes the variable w reverts, as the exp2 function does not support inputs greater than 192e18

In practice, this is what happened after switching from SD59x18 to UD60x18 - we started getting the following error when running the fuzz tests:

`PRBMath_UD60x18_Exp2_InputTooBig`

Thus, UD60x18 doesn't scale well for the mathematical complexity demanded by SablierV2LockupDynamic. We have to use SD59x18 due to the need to take the binary logarithm of values below 1e18, which requires in-flight signed numbers.

FYI, here are some examples of inputs that triggered reverts in testFuzz_Withdraw:

```
// args=[(0, [(0, 0, 0), (2475880078570760549798248447, 1844674407370955161 3, 436333)], 2,
// 0xF3408fb94475994988Ad732cC395EfEA04F124f1)]]
params.deposit = 0;
params.segments = new LockupDynamic.Segment[](2);
params.segments[0] = LockupDynamic.Segment({ amount: 0, exponent: ud2x18(0), milestone: 0 });
params.segments[1] = LockupDynamic.Segment({
    amount: 2_475_880_078_570_760_549_798_248_447,
    exponent: ud2x18(18_446_744_073_709_551_613),
    milestone: 436_333
});
params.timeWarp = 2;
params.to = 0xF3408fb94475994988Ad732cC395EfEA04F124f1;
```

These values are relatively large (the exponent is nearly MAX_UD2x18), but this doesn't change the fact that multiplications are hard to control, and switching to UD60x18 would have required us to severely constrain the design space of dynamic streams. Some limitations would have been awkward, such as prohibiting long-dated streams because they lead to tiny elapsedTimePercentage values.

Cantina: Acknowledged.

3.4.4 No scaling when casting numbers into `UD60x18` and `SD59x18`

Severity: Low Risk

Context:

- SablierV2LockupDynamic.sol#L319-L320

Description: When casting numbers into the `SD59x18` and `UD60x18` custom types the number of decimals isn't accounted for.

```
SD59x18 currentSegmentAmount = _streams[streamId].segments[index - 1].amount.intoSD59x18();
SD59x18 currentSegmentExponent = _streams[streamId].segments[index - 1].exponent.intoSD59x18();
```

For example in this case the amount is the amount of the underlying ERC20 which can have any number of decimals. DAI has 18, USDC has 6. Without scaling amounts of DAI and USDC would not result in the same SD59x18 representation; e.g. 1 USDC would be represented as 0.000000000001000000 and 1 DAI would be represented as 1.000000000000000000.

In the current code this does not represent a problem as there is no mix of amounts with different decimals used and the only place that uses the `pow` function has a percentage as base which is correctly represented with 18 decimals because of the division of two numbers with equal number of decimals (and a multiplication by 1e18 in the `div` function) It is unclear if this was done as an optimization as there is no inline comment about this.

Recommendation: Either as a best practice do scale according to the decimals of the amount being cast or add a comment indicating the optimization to the casual reader.

Sablier: Changes applied in PR 415.

Cantina: Confirmed.

3.4.5 ERC721 `tokenUri()` should revert on non existing `streamId`.

Severity: Low Risk

Context:

- SablierV2LockupDynamic.sol#L235-L237
- SablierV2LockupLinear.sol#L237-L239

Description: EIP721 defines the optional ERC721Metadata extension including the `tokenURI()` function. Although the metadata extension is optional, the EIP states the `tokenURI()` should revert when given a `tokenId` that does not exist.

```
/// @title ERC-721 Non-Fungible Token Standard, optional metadata extension
/// @dev See https://eips.ethereum.org/EIPS/eip-721
///  Note: the ERC-165 identifier for this interface is 0x5b5e139f.
interface ERC721Metadata /* is ERC721 */ {
    /// @notice A descriptive name for a collection of NFTs in this contract
    function name() external view returns (string _name);

    /// @notice An abbreviated name for NFTs in this contract
    function symbol() external view returns (string _symbol);

    /// @notice A distinct Uniform Resource Identifier (URI) for a given asset.
    /// @dev Throws if `_tokenId` is not a valid NFT. URIs are defined in RFC
    ///  3986. The URI may point to a JSON file that conforms to the "ERC721
    ///  Metadata JSON Schema".
    function tokenURI(uint256 _tokenId) external view returns (string);
}
```

The `SablierV2LockupDynamic` and `SablierV2Linear` contracts define the `tokenURI()` and forward the call to the `SablierV2NFTDescriptor` contract to resolve the `streamId` into a URI.

```
function tokenURI(uint256 streamId) public view override(IERC721Metadata, ERC721) returns (string memory uri) {
    uri = _nftDescriptor.tokenURI(this, streamId);
}
```

As the `SablierV2NFTDescriptor` contract (which is currently unimplemented and out of scope for this audit) will have no idea which `streamId` exist or not, the logical place to check existence is in the `SablierV2LockupDynamic` and `SablierV2Linear` contracts before the `SablierV2NFTDescriptor` contract is called and revert if needed.

Recommendation: Check that the `streamId` exist before calling the `SablierV2NFTDescriptor` and revert if the `streamId` does not exist.

Sablier: We have implemented a fix for this issue in PR 403.

Cantina: Fixed.

3.4.6 Stream creation can be front run by governance increasing the protocol fee.

Severity: Low Risk

Context:

- SablierV2Comptroller.sol#L74-L86
- SablierV2Comptroller.sol#L89-L9
- SablierV2LockupDynamic.sol#L281-L285
- SablierV2LockupLinear.sol#L290-L294
- SablierV2Base.sol#L80-L83

Description: The `SablierV2Comptroller` contract's `setProtocolFee()` function immediately takes effect and thus can be used by governance to front run anyone creating a stream. The same can happen with the `setComptroller()` in the `SablierV2Base` contract.

As there is no parameter for setting an acceptable maximum protocol fee when creating a stream, the user is forced to accept the new protocol fee and as it is payed in full immediately a cancel won't recuperate the protocol fee.

Note that this is not restricted to malicious intent of governance. The mere coincidence of creating a stream right after the transaction for changing the protocol fee or Comptroller contract is included in the chain would force a fee the user was not expecting.

Recommendation: Add a `maxAcceptableProtolFee` parameter to the create stream functions so a user can specify the current fee to avoid being front run.

Sablier: Acknowledged.

Cantina: Acknowledged.

3.4.7 Getter functions do not revert on non existing `streamId`

Severity: Low Risk

Context:

- SablierV2LockupDynamic.sol#L99-L249
- SablierV2LockupLinear.sol#L84-L251

Description: Following getter functions do not revert on non existing `streamIds`.

- getAsset()
- getCliffTime()
- getDepositAmount()
- getEndTime()
- getRange()
- getRecipient()
- getSegments()
- getSender()
- getStartTime()
- getStatus()
- getStream()
- getWithdrawnAmount()
- isCancelable()
- returnableAmountOf()
- streamedAmountOf()
- withdrawableAmountOf() It is considered best practice to revert on invalid inputs. For example all ERC721 functions revert when given a `tokenId` that does not exist. This is especially relevant when integrating with other Defi contracts as those contracts may assume reverts on invalid inputs.

Recommendation: Revert on calls getting information on a `streamId` when that `streamId` does not exist.

Sablier: We implemented a fix for this issue in this PR 412. A couple of notes:

- It doesn't make sense to revert in `getStatus`, since this is the very function used to check for the existence of a stream. If the status of a stream is NULL, it means the input id is invalid. Thus, we kept this particular function unchanged.
- All others getters revert now.
- `getRecipient` reverts with the revert reason string "ERC721: invalid token ID", because the NFT can be burned and in this case, the NFT does not exist anymore.
- All other getters revert with a new custom error `SablierV2Lockup_NullStream`.
- Reverting invalid inputs bears implications for the `cancelMultiple` function. See this issue.

Cantina: Verified.

3.4.8 Use `_safeMint()` rather than `_mint()`

Severity: Low Risk

Context:

- SablierV2LockupDynamic.sol#L512
- SablierV2LockupLinear.sol#L431)

Description: The `SablierV2LockupDynamic` and `SablierV2Linear` contract make us of the standard `_mint()` function when minting the ERC721 to the receiver.

```
function _createWithMilestones(LockupDynamic.CreateWithMilestones memory params) {
    ...
    _mint({ to: params.recipient, tokenId: streamId });
}
```

This allows contracts that are not set up to support ERC721 tokens to receive the tokens, which can result in the NFT being stuck in the contract.

OpenZeppelin's `_mint()` function however does not check the `onERC721Received()` (which is included by contracts to indicate their ability to properly handle ERC721 tokens) whereas the `_safeMint()` does.

Recommendation: Use `_safeMint()` over `_mint()` so that the creation of the stream reverts when the recipient does not support ERC721.

Sablier: We will mitigate this issue at the UI level by showing a warning to senders that will say something along the lines of "make sure the recipient is an address that is able to interact with the functions of the Sablier contract". Issue 22.

Cantina: Acknowledged.

3.4.9 Protocol fee is checked on stream creation instead of when set

Severity: Low Risk

Context: SablierV2Comptroller.sol#L74-86

Description: When protocol fees are set by the admin using the `setProtocolFee()` function, there are no checks on whether it is less than `MAX_FEE`:

```
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
    });
}
```

Instead, this comparison is checked in `Helpers::checkAndCalculateFees()`:

```
if (protocolFee.gt(maxFee)) {
    revert Errors.SablierV2Lockup_ProtocolFeeTooHigh(protocolFee, maxFee);
}
```

Since the `MAX_FEE` value is immutable, it is possible to check each newly set protocol fee once, in the setter, and then trust that it will always be under `MAX_FEE` when used. This would be more ideal for two reasons:

1) Instead of using gas as each stream is created to perform this check, we would be able to perform it just once when fees are set.
2) In the event that a fee is set too high in the current implementation, users will be locked from creating streams with this asset. Although this is unlikely and could quickly be corrected by the Sablier team, the better behavior would be to catch an incorrect setting as it was being set.

Recommendation: Move the check that `protocolFee < MAX_FEE` into the `setProtocolFee()` function, and remove this check from `Helpers::checkAndCalculateFees()`.

Sablier: From a technical standpoint, this approach may be sound; however, its implementation would compromise the protocol's trustlessness. By confining this check solely to the comptroller, governance would be granted the ability to set MAX_FEE to any value, due to the existence of the setComptroller function. Consequently, users would lack a hard-coded guarantee that the fees charged by Sablier will not exceed a certain percentage, even in the most unfavorable situations.

It is also relevant to note that relocating the max fee to the comptroller would contradict other identified findings, specifically those mentioned in reports #3 and #15.

Of course, in the paragraphs above, I assumed you were referring to storing the MAX_FEE in the comptroller. However, in principle, it is also possible to query the MAX_FEE from the streaming contracts. But we don't want this, because:

1. The max fee may differ across streaming contracts.
2. The flow of communication is always from the streaming contracts to the comptroller, not the other way around (we expect to ship more streaming contracts in the future)
3. The reference structure would be awkward; the streaming contracts would reference the comptroller, and the comptroller would reference one (or more) streaming contracts. This design is prone to errors, as it relies on governance to keep the references in sync.

Therefore, we will keep the implementation as is (i.e. keep the check-in Helpers::checkAndCalculateFees).

Cantina: We discussed this in more detail on a call and agree that, given the current architecture and the fact that MAX_FEE is stored on the upgradeable controller, the best approach to ensure trustlessness without major changes is to leave this as-is.

3.4.10 Streams can be started with an end time in the past

Severity: Low Risk

Context: SablierV2LockupLinear.sol#L395-L421

Description: When a new stream is created, there are no checks on the `startTime` or `endTime` parameters relative to the current time. All checks are only relative to each other.

This opens up a dangerous avenue for user error, where incorrectly entering an `endTime` could result in a stream that is immediately fully vested.

In such a situation, even canceling would not help, as the full value of the stream would be transferred to the recipient.

In order to help prevent this kind of user error, it is recommended to ensure that `endTime > block.timestamp`.

Recommendation: In the `Helpers.sol` functions that check the time params, include the following check:

In `checkCreateLinearParams()`:

```
if (block.timestamp >= range.end) {
    revert Errors.SablierV2LockupLinear_CurrentTimeNotLessThanEndTime(block.timestamp, range.end);
}
```

In `_checkSegments()`:

```
if (block.timestamp >= previousMilestone) {
    revert Errors.SablierV2LockupDynamic_CurrentTimeNotLessThanEndTime(block.timestamp, range.end);
}
```

Sablier: We have implemented a fix for this issue in PR 396.

Cantina: Confirmed, issue resolved as recommended and test updates look great too.

3.4.11 Critical admin transfers should be two step process

Severity: Low Risk

Context: Adminable.sol#L34-40

Description: The Admin role is critical to the functioning of the protocol, as this is the only address able to claim protocol revenues (in addition to setting fees and allow flash assets). If this role were to be incorrectly transferred, all protocol revenues would become permanently stuck.

The contract allows this role to be transferred in one transaction, which leaves open the possibility that it could be transferred to an incorrect address.

It is recommended to implement a two-step transfer process for such important transfers, to ensure that you own the asset on the other end to successfully claim ownership.

Recommendation: Change the transfer of ownership to a two step process where:

1) The current admin calls `setPendingAdmin()` to set a new `pendingAdmin` value in storage.
2) The `pendingAdmin` calls `acceptAdminRole()` to zero out the `pendingAdmin` value and set themselves as `admin`.

Sablier: We will leave this as it is currently implemented (a single-step process), because of three reasons:

1. The admin will be set to a multisig, which means that we will implicitly have a multiple-step process for submitting txs. We will always verify one another's transactions before submitting them.
2. We don't expect to turn on the protocol fees any time soon.
3. By the time we activate the protocol fees, we may have a governance module in place. This module will follow the industry practices and implement a time delay for executing proposals.

Cantina: Discussed in more detail on a call and it sounds like they are comfortable with their protections living before the transaction is submitted rather than after. We still believe two step transfers are safer, but agree that the safeguards they describe should be sufficient to avoid any issue.

3.5 Gas Optimization

3.5.1 `returnableAmountOf()` can be refactored to save gas with implicit return

Severity: Gas Optimization

Context: SablierV2LockupLinear.sol#L173-184

Description: The `returnableAmountOf()` function calculates the amount of the initial deposit that has not been streamed for cancellations:

```
function returnableAmountOf(uint256 streamId) external view returns (uint128 returnableAmount) {
    // When the stream is not active, return zero.
    if (_streams[streamId].status != Lockup.Status.ACTIVE) {
        return 0;
    }

    // No need for an assertion here, since {streamedAmountOf} checks that the deposit amount is greater
    // than or equal to the streamed amount.
    unchecked {
        returnableAmount = _streams[streamId].amounts.deposit - streamedAmountOf(streamId);
    }
}
```

In the case that the status is not active, early returning zero uses additional unneeded gas, since the `returnableAmount` return value is already set to 0 by default.

Instead, we can only perform the calculation in the event that the status is active, and defaults to returning zero otherwise.

Here's a simplified version of the function to demonstrate gas savings:

```
mapping(uint256 => bool) private _streamStatus;
mapping(uint256 => uint128) private _streamDeposits;
mapping(uint256 => uint128) private _streamWithdrawals;
```

```
function testImplicitReturnGas() public {
    _streamStatus[1] = true;
    _streamDeposits[1] = 100;
    _streamWithdrawals[1] = 50;

    _streamStatus[2] = false;
    _streamDeposits[2] = 100;
    _streamWithdrawals[2] = 50;

    uint t11a = gasleft();
    uint x = returnableTestV1(1);
    uint t11b = gasleft();

    uint t12a = gasleft();
    uint z = returnableTestV1(2);
    uint t12b = gasleft();

    uint t21a = gasleft();
    uint y = returnableTestV2(1);
    uint t21b = gasleft();

    uint t22a = gasleft();
    uint w = returnableTestV2(2);
    uint t22b = gasleft();

    // Gas savings occur whenever we are inactive:
    console.log("in the INACTIVE case", t12a - t12b, t22a - t22b);

    // Gas isn't harmed (actually improves by 1 gas) in the gas where we are active:
    console.log("in the ACTIVE case", t11a - t11b, t21a - t21b);
}

function returnableTestV1(uint256 streamId) internal view returns (uint128 returnableAmount) {
    // When the stream is not active, return zero.
    if (_streamStatus[streamId] == false) {
        return 0;
    }

    // No need for an assertion here, since {streamedAmountOf} checks that the deposit amount is greater
    // than or equal to the streamed amount.
    unchecked {
        returnableAmount = _streamDeposits[streamId] - _streamWithdrawals[streamId];
    }
}

function returnableTestV2(uint256 streamId) internal view returns (uint128 returnableAmount) {
    // When the stream is not active, return zero.
    if (_streamStatus[streamId] == true) {
        unchecked {
            returnableAmount = _streamDeposits[streamId] - _streamWithdrawals[streamId];
        }
    }
}
```

```
Logs:
  in the INACTIVE case 330 311
  in the ACTIVE case 754 753
```

Recommendation: Refactor the function to take advantage of the implicit return, similar to `returnableTestV2()` above.

Sablier: Thanks for the gas comparison. We will refactor the returnableAmountOf function as you suggest. While working on the refactor, I realized that we could employ the same trick in the isCancelable function. We implemented this gas optimization in PR 412."

Cantina: Confirmed.

3.5.2 `cancelMultiple()` checks status twice

Severity: Gas Optimization

Context: SablierV2Lockup.sol#L140

Description: When canceling multiple streams using the `cancelMultiple()` function, we loop over the streams and perform the following check and action:

```
if (getStatus(streamId) == Lockup.Status.ACTIVE && isCancelable(streamId)) {
    _cancel(streamId);
}
```

There is no need to check the lockup status, as the exact same check is made within `isCancelable()`:

```
function isCancelable(uint256 streamId)
    public
    view
    override(ISablierV2Lockup, SablierV2Lockup)
    returns (bool result)
{
    // A null stream does not exist, and a canceled or depleted stream cannot be canceled anymore.
    if (_streams[streamId].status != Lockup.Status.ACTIVE) {
        return false;
    }
    result = _streams[streamId].isCancelable;
}
```

Recommendation: Remove the extra check:

```
- if (getStatus(streamId) == Lockup.Status.ACTIVE && isCancelable(streamId)) {
+ if (isCancelable(streamId)) {
      _cancel(streamId);
  }
```

Sablier: Nice catch. We have implemented a fix for this issue in PR 412.

Cantina: Confirmed, the refactored version that performs checks in the separated `cancel()` and `renounce()` functions resolves this issue.

3.6 Informational

3.6.1 `_isApprovedOrOwner()` does not override OZ ERC721 version.

Severity: Informational

Context:

- SablierV2LockupDynamic.sol#L375-L378
- SablierV2LockupLinear.sol#L305-L308

Description: The `SablierV2Lockup` contract defines a `_isApprovedOrOwner(uint256,address)` function that has a different function signature than the `_isApprovedOrOwner(address,uint256)` defined in the OpenZeppelin ERC721 contract. To the casual reader however this can easily be misread as being an override of the OpenZeppelin function.

```
function _isApprovedOrOwner(address spender, uint256 tokenId) internal view virtual returns (bool) {
    address owner = ERC721.ownerOf(tokenId);
    return (spender == owner || isApprovedForAll(owner, spender) || getApproved(tokenId) == spender);
}
```

Recommendation: Consider renaming the function in order to clearly distinguish it from the OZ version.

Sablier: We have implemented the recommendation in this PR 408 .

Cantina: Confirmed.

3.6.2 Regulatory risk

Severity: Informational

Context: General observation

Description: Some tokens have features to comply with regulations, most notably Circle maintains a blocklist to comply with OFAC sanctions in their USDC contracts. When the recipient/owner of a stream is an address that is on USDC's blocklist they can withdraw the USDC to another address and bypass the blocklist (for future withdrawals, not past ones). This could lead Circle to add the Sablier contracts to the USDC blocklist thereby blocking all user's streams and locking the USDC of all the streams in the Sablier contracts. The likelihood of this happening is unknown and as such we have designated it as low but we feel it's prudent to warn about this risk as it would probably be an existential risk for Sablier.

Recommendation: Consider creating separate (minimal proxy) vaults for each stream where the deposits for the streams will be held. This would limit the impact of such an action by an underlying token as it would only lock the tokens for that stream. Additionally each stream could have 2 vaults, one for the sender and one for the recipient as to not lock the funds of the sender when the recipient is the target of the regulatory action (or the inverse). This would still allow the non-target to withdraw their funds even if the target vault has been blocked. The extra benefit is that it allows the use of a sender NFT which controls the sender's vault and makes it possible to transfer those access rights to other addresses and even for use in Defi contracts which is currently impossible (for the sender although they're locking their funds in the stream even more so than the recipient). Note however that this would complicate the flash loan function as the funds would not be concentrated in one contract anymore.

Sablier: Thank you for deciding to post this issue. It is an interesting risk, for sure!

> *This could lead Circle to add the Sablier contracts to the USDC blocklist* As far as I know, there is no precedent for Circle blacklisting contracts where USDC is pooled. If they did this, they would also blacklist many innocent users, significantly hurting their business. We expect Sablier to have many more non-blacklisted users than blacklisted users. *Consider creating separate (minimal proxy) vaults for each stream* While this approach is technically feasible, it would lead to a massive refactoring of our contracts. We would basically have to go back to the whiteboard. We also like the monolith design better - all else being equal, it involves fewer security assumptions.

The other possibility is to implement an `isRecipientBlacklisted` function in the comptroller. However, this would have many undesirable consequences, such as giving discretionary power to Sablier governance over who can use the protocol and who can't.

Cantina: Acknowledged.