Superfluid Scheduler Security Review

Review Resources: Superfluid Docs

Reviewers: engn33r

Table of Contents

Review Summary

Superfluid Scheduler

The contracts of the Superfluid Platform Repo were reviewed over 14 days. The code review was performed between November 14 and 28, 2022. There were 29 man hours of effort spent on this review, which includes time spent on the code review process and discussions with the development team. The repository was under active development during the review, but the review was limited to the code at commit 0b8218856863a2db4ec78daebafd00f4c695112e.

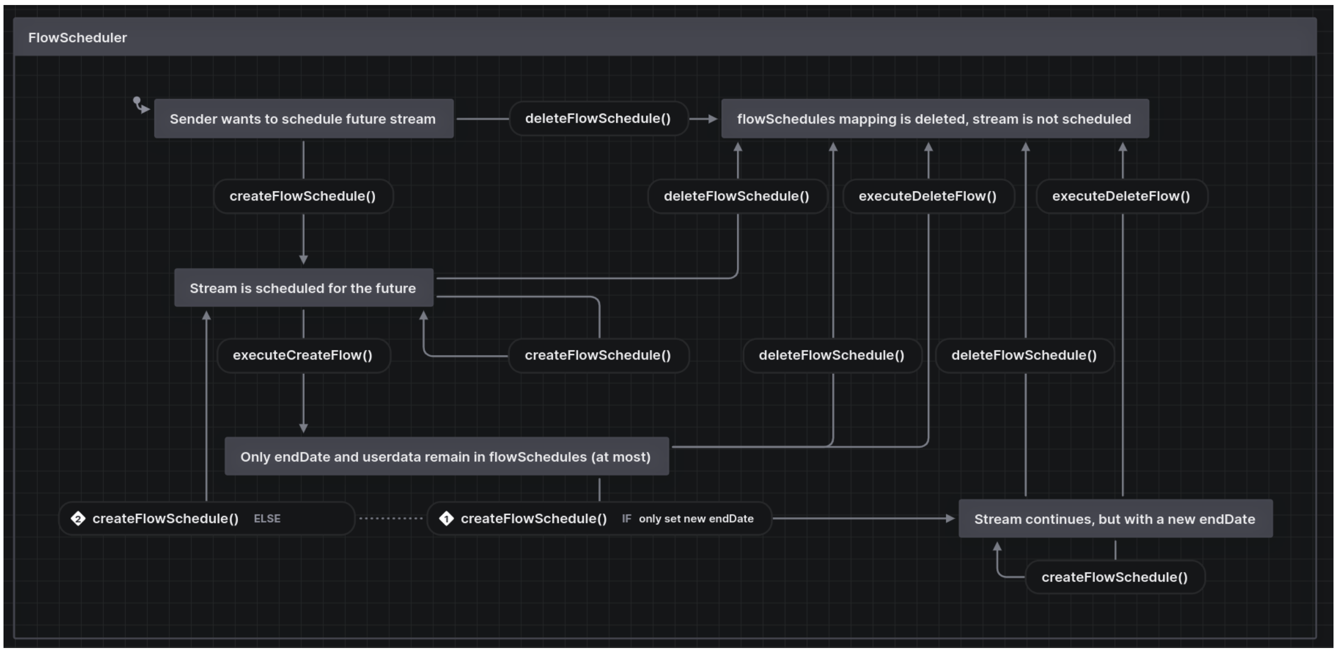
Scope

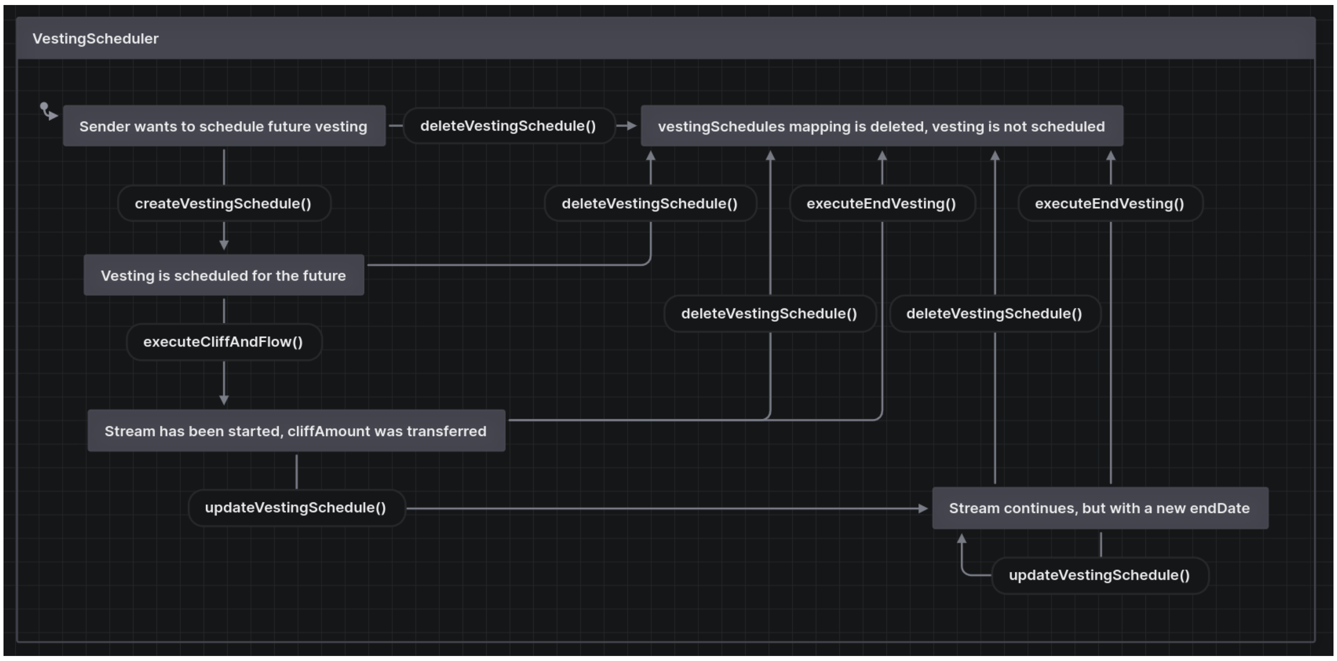
The scope of the review consisted of the following contracts at the specific commit. Other contracts of the Superfluid protocol, including those already in use to handle standard Superfluid streams, were not in scope of this review:

- FlowScheduler.sol
- VestingScheduler.sol

- IFlowScheduler.sol
- IVestingScheduler.sol

These contracts use a CRUD (create, read, update, delete) design pattern. The Scheduler contracts are “dumb” Superfluid Super Apps (meaning there are no Super App agreement callbacks). Calls are normally performed by the main Superfluid host contract but can also also be directly performed by the sender. An overview of the expected scheduling callflow for each contract is shown below.


FlowScheduler
Sender wants to schedule future stream
deleteFlowSchedule()
flowSchedules mapping is deleted, stream is not scheduled
createFlowSchedule()
deleteFlowSchedule()
executeDeleteFlow()
executeDeleteFlow()
Stream is scheduled for the future
executeCreateFlow()
createFlowSchedule()
deleteFlowSchedule()
deleteFlowSchedule()
Only endDate and userdata remain in flowSchedules (at most)
createFlowSchedule() ELSE
createFlowSchedule() IF only set new endDate
Stream continues, but with a new endDate
createFlowSchedule()



The unique feature that the Scheduler contracts introduce to the Superfluid protocol is allowing a flow or vesting to be scheduled for a future date, without requiring user intervention to start the flow in the future. This is enabled by Superfluid’s backend system, which tracks scheduled flows and vestings to call the execute function when the future point in time has arrived. This backend system works in a similar way to the Keep3r network or Gelato network. When a flow start time is reached, the backend keepers call the existing Superfluid smart contracts to schedule the stream at the appropriate time. The same happens when the scheduled end date is reached, and the backend keepers can end the stream at the appropriate time.

This review was a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

The authors of this report make no warranties regarding the security of the code and do not warrant that the code is free from defects. The authors of this report do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Superfluid and users of the contracts agree to use the code at their own risk.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements,
- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

Critical Findings

None.

High Findings

None.

Medium Findings

1. Medium - Limitation on parallel flows

There is a limitation in VestingScheduler that only one vesting flow with a specific combination of superToken, sender, and receiver is allowed. If a second flow is created with the same superToken, sender, and receiver, the `createVestingSchedule()` function will revert.

Technical Details

The custom ScheduleAlreadyExists error in VestingScheduler prevents duplicate two streams with matching `hashConfig` values. While it might be unlikely that a user would set up two vesting streams to the same receiver with the same currency, this line of code prevents the user from using Superfluid in that case, effectively creating a denial of service for that user's goal. FlowScheduler follows a similar design, but an attempt to create a parallel flow with the same combination of superToken, sender, and receiver as an existing flow will not revert but rather overwrite the existing flow. This difference between VestingScheduler and FlowScheduler is visible in the state machine diagram above.

Impact

Medium. A user would be unable to schedule a specific Superfluid vesting flow in certain cases.

Recommendation

The key problem is that the values used in the mapping for parallel flows are not unique, imposing an inherent limitation. Including a unique value in the `hashConfig` calculation would allow for the creation of multiple streams. For example, the block number or block timestamp are non-repeating values. An incrementing variable in the Scheduler contracts is another option to make sure there is no reuse of existing values.

Developer Response
Acknowledged, we didn't want to include additional complexity at the initial phase but adding this feature will be considered for future versions.

2. Medium - Lack of `flowDelayCompensation` in FlowScheduler can undercompensate on late flow execution

If `executeCreateFlow()` is called significantly after `startDate`, for example closer to `startMaxDelay` than expected, the flow will start late. The time differential between `block.timestamp` and the actual start time of the flow equates to value not received by the receiver. Ideally the receiver should receive this payment up-front at the start of the flow.

Technical Details
VestingScheduler has a `flowDelayCompensation` compensation amount at the time of vesting execution but FlowScheduler has no comparable calculation. In the interest of using the flow start time to accurately determine the amount of value the receiver receives, compensation should be sent to the receiver at the start of the flow.

Impact
Medium. The value transferred during the flow may not align with the expectations of the sender in the case that the flow start is delayed.

Recommendation
Implement a variable similar to `flowDelayCompensation` in FlowScheduler to compensate the receiver for delayed flows at the start of the flow.

Developer Response
This difference in FlowScheduler and VestingScheduler is intentional. FlowScheduler is designed for simpler cases, where a scheduling a flow start and stop is the goal, while VestingScheduler includes additional features for accuracy of the overall payment amount. Avoiding `flowDelayCompensation` in FlowScheduler provides some protection against a sender getting liquidated in the case where the sender has insufficient tokens.

3. Medium - Reverting on flow state change when `sender == msg.sender`

The `_getSender()` internal function contains logic for when the function caller is the same as the stream sender, meaning `sender == msg.sender` should be permitted. But the external `createFlowByOperator()` and `deleteFlowByOperator()` functions contain logic to revert in this case, requiring custom handling that is already found in another Superfluid contract.

Technical Details

One contract in the existing Superfluid repository that calls the external CFA `createFlowByOperator()` function is CFAv1Forwarder, and it contains special logic in the case where `sender == msg.sender`. This is because of a revert condition in `createFlowByOperator()` when `currentContext.msgSender == sender`. Similar branching logic in the `deleteFlow` function of CFAv1Forward, likely because `deleteFlowByOperator()` has a similar revert condition, although it is implemented differently and less obvious. To prevent this revert and allow the sender to call the functions `executeCreateFlow()`, `executeCliffAndFlow()`, `executeDeleteFlow()`, and `executeEndVesting()`, borrow the same logic as CFAv1Forwarder.

Impact

Medium. This function is normally called by a keeper and not the sender, but it should be possible for the sender to call this function because the sender is permitted to call all the other functions in the Scheduler contracts.

Recommendation

Borrow the logic found in CFAv1Forwarder so that `createFlow()` is called instead of `createFlowByOperator()` when `sender == msg.sender`. The tests for the two Scheduler contracts should test three cases for calling all Scheduler functions using `msg.sender == sender`, `msg.sender == address(cfaV1.host)`, `msg.sender != sender && msg.sender != address(cfaV1.host)`. The first two cases should be permitted while the last case should revert in the functions protected by `_getSender()`. Ideally the last of these three cases should be fuzzed with foundry for additional security.

Developer Response

Inaccurate finding. Superfluid ACL permissions are set to the FlowScheduler or VestingScheduler contract address. Any user calling it will always use the contract address. The user can execute the operation even if is sender of stream using the *byOperator functions because it will always use the contract address.

Low Findings

1. Low - Poorly chosen `startMaxDelay` can make flow unscheduleable

There is no check in `createFlowSchedule()` for a minimum value of `startMaxDelay`. A zero or very small value of `startMaxDelay` can make the flow impossible to schedule, even assuming zero latency.

Technical Details

The logic to start a flow, found in `executeCreateFlow()`, is:

```
if (
            schedule.startDate == 0 ||
            schedule.startDate > block.timestamp ||
            schedule.startDate + schedule.startMaxDelay < block.timestamp
            ) {
            revert TimeWindowInvalid();
}
```

If `startMaxDelay` is zero, then it is guaranteed that the logical statement `schedule.startDate > block.timestamp || schedule.startDate < block.timestamp` will evaluate to true unless `startDate == block.timestamp`, which is unlikely given the 12-second block times (roughly 1/12 chance or 8.3%). This 8.3% chance of starting assumes no latency and perfect timing from the backend. In any other scenario, a `startMaxDelay` of zero will cause the call to create the flow will revert with the `TimeWindowInvalid` error.

Impact

Low. An unscheduleable flow can be scheduled even though it will never get started.

Recommendation

Set a minimum `startMaxDelay` value, for example 1 hour, and add a check for this value in `createFlowSchedule()`. This is similar to what is done in VestingScheduler, where `START_DATE_VALID_AFTER` is set to a hardcoded value of 3 days. Setting of this value depends on the backend infrastructure and therefore is best determined by the Superfluid dev team.

Developer Response

Acknowledged, won't fix. The scheduler contracts do not give any guarantee about a scheduled flow or vesting being scheduleable. There are other edge cases where the flow or vesting cannot be started, such as a scenario where the sender has insufficient tokens for the stream to start. There is no benefit to setting limits on `startMaxDelay` because it would only change the probability that the flow could be started. There are other edge cases that cannot easily be prevented.

2. Low - A large `cliffAmount` value can cause revert

An overflow can occur in `executeCliffAndFlow()` if the value of `cliffAmount` is `type(uint256).max` (or a similarly large value). The revert is due to an overflow because adding any non-zero value to `type(uint256).max` will trigger an overflow that will be caught by SafeMath.

Technical Details

This line of VestingScheduler sums `cliffAmount` and `flowDelayCompensation`. If `cliffAmount` is already the max uint256 value, then adding any non-zero `flowDelayCompensation` value will cause an overflow, which will revert due to the SafeMath in Solidity >=0.8.0. This will cause the entire `executeCliffAndFlow()` call to revert, meaning the scheduled vesting cannot be started unless `flowDelayCompensation` is exactly zero, which is very hard to do.

Impact

Low. The sender/scheduler has ultimate control over the flows they schedule, so the only risk would be difficulties created for the Superfluid backend, like a denial of service condition that wastes resources.

Recommendation

Use a uint128 for the `cliffAmount`. If a smaller sized uint is used, it may even save gas depending on variable packing.

Developer Response

Acknowledged. Similar to the previous finding, `executeCliffAndFlow()` can fail for many reasons so adding an extra check would not add much value.

3. Low - Calling `executeEndVesting()` without `executeCliffAndFlow()` skips `cliffAmount` transfer

VestingScheduler is designed for `executeCliffAndFlow()` to always be called before `executeEndVesting()`. But if `executeCliffAndFlow()` is not called during the

3-day `START_DATE_VALID_AFTER` time window, `executeEndVesting()` can be called without the receiver receiving their `cliffAmount` payment. This would allow the flow to end without error, but would leave the receiver short of the cliff payment so the sender retains more value than expected.

Technical Details

The FlowScheduler and VestingScheduler contracts are intended to schedule future streams. The backend keeper system that triggers the on-chain events of starting and stopping flows cannot be guaranteed to work with 100% success rate unless the keeper incentive structure and uptime can be guaranteed to work 100% of the time. With this in mind, there is a small chance that any keeper-triggered function call does not happen and the stream does not start, stop, or get executed as planned. These edge cases are considered in the stream liquidation process, where different layers of keepers (particians, plebs, and pirates) liquidate problem streams. One special case of the scenario where keepers don’t perform their job is when an intermediate step in the process of scheduling a flow is skipped, but the remaining steps of the flow succeed.

While `createVestingSchedule()` contains a check to make sure that `executeEndVesting()` is not possible to call before `executeCliffAndFlow()`, `executeEndVesting()` does not include any check to confirm that `executeCliffAndFlow()` was previously called. Because the sequence of allowed function calls is designed to be strict, it is necessary to enforce the proper call sequence of these functions. Even though under normal operating conditions a 3-day `START_DATE_VALID_AFTER` time window should be enough for keepers to call `executeCliffAndFlow()`, unexpected edge cases (issues with keepers, issues with high gas on the blockchain, unforeseen circumstances, etc.) could cause this 3-day time window to pass without `executeCliffAndFlow()` getting called for this vesting. `executeEndVesting()` could then be called in order to end the flow even though the `cliffAmount` is non-zero and was never sent to the receiver.

Impact

Low. There is a very small chance that the receiver may not receive the value expected from the scheduled vesting, but this is an inherent risk when involving a keeper system to trigger on-chain events.

Recommendation

Integrating with a keeper solution that has the best incentive structures to maintain the highest uptime, availability, and keeper responsiveness is one way to increase the probability that the on-chain function calls get triggered.

Add logic to add `cliffAmount` to the `earlyEndCompensation` sum that is transferred to the receiver if `cliffAmount` is not zero. The new logic could look similar to the code below, but whether the `cliffAmount` transfer should remain in a try-catch is unclear as this choice depends on the preferred failure mode if the sender does not have enough tokens for the cliff payment.

```
if (earlyEndCompensation + schedule.cliffAmount > 0) {
            // try-catch this because if the account does not have tokens for
earlyEndCompensation we should delete the flow anyway.
            try superToken.transferFrom(sender, receiver, earlyEndCompensation +
schedule.cliffAmount)
            {} catch {
                        didCompensationFail = true;
            }
}
```

Developer Response

Acknowledged, won't fix. If this case happens, the sender should delete the vesting and start a new one with the proper values. Sending `cliffAmount` in `executeEndVesting()` breaks the expectation about the receipt of `cliffAmount` itself. The receiver would be denied the `cliffAmount` at the agreed upon time (beginning of the stream) and it is postponed potentially years down the line.

4. Low - `updateVestingSchedule()` may be called at any time after `executeCliffAndFlow()`

A vesting may have its `endDate` value updated at the same time that the stream is eligible to be ended by calling `executeEndVesting()`. While this possibility is permitted with the current contract logic, it may be preferable to update the logic in the `updateVestingSchedule()` contract to disallow this possible race condition.

The same issue exists with FlowScheduler too. FlowScheduler does not have an update function, but `createFlowSchedule()` can be called to change only the `endDate` after `executeCreateFlow()` has started the flow and the current block.timestamp value permits the calling of `executeDeleteFlow()`.

Technical Details

`updateVestingSchedule()` can be called at any time after `executeCliffAndFlow()` has been called as long as `schedule.endDate != 0` (meaning `executeEndVesting()` has not been called yet). This means that `updateVestingSchedule()` can be called in these unexpected situations:

1. After a stream has been ended using a method outside of the Scheduler contract, the `endDate` will remain in the mapping because calling `executeEndVesting()` will revert on the `cfaV1.deleteFlowByOperator()` call.
2. After the keeper initiates a call to `executeCliffAndFlow()`, the sender can frontrun this transaction to change the `endDate` so that it does not meet the criteria to call `executeEndVesting()`. This could result in a scenario where the backend team is troubleshooting why a keeper called `executeEndVesting()` to end a vesting that does not appear (at the current time) to qualify for ending based on `endDate`.
3. In the most extreme case, the two issues above can be combined to confuse or potentially disrupt the handling of keepers. The sender would monitor the mempool for calls to `executeEndVesting()`, then either frontrun them to end the stream before the `executeEndVesting()` call is completed OR end the stream through an alternate method. One `executeEndVesting()` is uncallable on this stream due to the revert on the `cfaV1.deleteFlowByOperator()` call, the sender can move the `endDate` value far into the future, say to `type(uint32).max`, and repeat this process many times to force the contract to continue storing a lot of state variable storage in the mapping.

The same scenario is possible in FlowScheduler. The only difference is that `createFlowSchedule()` would be used instead of an update function, and the existing mapping would be overwritten. Zero values are permitted for all the function arguments that `executeCreateFlow()` deletes so that the only value that is overwritten is the `endDate` value.

Impact

Low. It is unclear whether the backend will handle such a race condition situation cleanly. Most likely the `updateVestingSchedule()` function should be revised to disallow this edge case.

Recommendation

To prevent the sender from calling `updateVestingSchedule()` at unexpected times, modify this line of code:

```
- if (endDate <= block.timestamp) revert TimeWindowInvalid();
+ if (endDate <= block.timestamp || schedule.endDate - END_DATE_VALID_BEFORE <=
block.timestamp) revert TimeWindowInvalid();
```

Add tests that better cover this case. For example, the existing `testCreateFlowScheduleWithZeroTimes` test in FlowScheduler.t.sol only tests the creation of a flow with `startDate == 0`, but there is no test to check if all variables that are set to zero in `executeCreateFlow()` (specifically `startAmount`, `startDate`, `startMaxDelay`, and `flowRate`) can be set to zero during the initial `createFlowSchedule()` flow scheduling. The case of scheduling a flow in FlowScheduler with a flowrate of zero might be a case to prevent in FlowScheduler's `createFlowSchedule()` because there are checks elsewhere in the existing Superfluid codebase where a flowrate of zero reverts.

Developer Response

This is by design, the sender should be able to change the vesting end date if the stream has started. At the end of the day, the sender has control over the stream they are sending, as they should.

Gas Savings Findings

1. Gas - Revert early when startDate == 0

If a flow is scheduled with `startDate == 0`, which is permitted according to the NatSpec in IFlowScheduler, it will later be impossible to start the flow. Instead of allowing unstartable flows to be scheduled, alter the logic to prevent this case.

Technical Details

If `startDate == 0`, `executeCreateFlow()` will revert on this line.

`startDate == 0` should revert even earlier, in `createFlowSchedule()`, to prevent this case from happening. Currently `createFlowSchedule()` only reverts with `startDate == 0` if other logic is also true.

Impact

Gas savings.

Recommendation

Modify `createFlowSchedule()`

```
if (startDate == 0) {
-          if ( startMaxDelay != 0 || endDate == 0) revert TimeWindowInvalid();
+          revert TimeWindowInvalid();
}
```

Then remove the `startDate == 0` check from `executeCreateFlow()` because this value will not be possible to store in the `flowSchedules` mapping.

```
if (
-          schedule.startDate == 0 ||
           schedule.startDate > block.timestamp ||
           schedule.startDate + schedule.startMaxDelay < block.timestamp
```

Lastly, update the NatSpec comments in IFlowScheduler to remove comments about the validity of a zero value for startDate.

Developer Response

`startDate == 0` is an acceptable value in `createFlowSchedule()` so this change does not make sense.

2. Gas - Remove `userData` function argument

The only purpose of the `userData` function argument is to revert if the function argument does not match the stored value. Instead, this function argument can be removed and the stored value trusted as the correct value.

Technical Details

The only time the `userData` function argument is used is in this validation check that reverts if the stored value does not match the function argument. The same is true in `executeDeleteFlow()`. The correct `userData` value can be viewed on the blockchain so requiring the user to provide it as a security check doesn't provide much value.

Impact

Gas savings.

Recommendation

Remove the `userData` function argument from `executeCreateFlow()` and `executeDeleteFlow()`. Then remove the two lines with userData checks that contain the `UserDataInvalid` error and remove the `UserDataInvalid` error from IFlowScheduler.sol.

Developer Response

`userData` is an additional field used when opening a CFA Stream that the receiver can decode to get more information. This happens on-chain and the third-party executing it should provide this data.

3. Gas - Use unchecked if no underflow risk

There is a subtraction operation that can use unchecked for gas savings.

Technical Details

One example of where unchecked can be applied in VestingScheduler

```
- uint256 earlyEndCompensation = schedule.endDate > block.timestamp ?
(schedule.endDate - block.timestamp) * uint96(schedule.flowRate) : 0;
+ unchecked { uint256 earlyEndCompensation = schedule.endDate > block.timestamp ?
(schedule.endDate - block.timestamp) * uint96(schedule.flowRate) : 0; }
```

Another example of where unchecked can be applied in VestingScheduler

```
- uint256 flowDelayCompensation = (block.timestamp - schedule.cliffAndFlowDate) *
uint96(schedule.flowRate);
+ unchecked { uint256 flowDelayCompensation = (block.timestamp -
schedule.cliffAndFlowDate) * uint96(schedule.flowRate); }
```

Impact

Gas savings.

Recommendation

Use unchecked if there is no overflow or underflow risk for gas savings.

Developer Response

There is no concrete evidence or fuzzing test case provided to show that the math will not overflow. No change necessary at this time.

4. Gas - Use != 0 for gas savings

Using `> 0` is gas inefficient compared to using `!= 0` when comparing a uint to zero. This improvement does not apply to int values, which can store values below zero.

Technical Details

About a dozen instances of checks including `> 0` are found throughout FlowScheduler and VestingScheduler.

Impact

Gas savings.

Recommendation

Replace `> 0` with `!= 0` to save gas.

Developer Response

Fixed in PR 77.

5. Gas - Redundant logic check repeats another check

There are three constant duration variables in VestingScheduler that cannot be changed, because they are constants. There is a redundant check involving these constants.

Technical Details

The first of the two checks here involving the constant duration variables is redundant, and is in fact a weaker (less strict) check than the second one.

The first check can be simplified as follows:

```
cliffAndFlowDate + START_DATE_VALID_AFTER >= endDate - END_DATE_VALID_BEFORE
cliffAndFlowDate + 3 days >= endDate - 1 day
4 days >= endDate - cliffAndFlowDate
```

The second check can be simplified as follows:

```
endDate - cliffAndFlowDate < MIN_VESTING_DURATION
endDate - cliffAndFlowDate < 7 days
endDate < cliffAndFlowDate + 7 days
7 days > endDate - cliffAndFlowDate
```

As seen above, the second check validates that the difference between `endDate` and `cliffAndFlowDate` must be greater than or equal to 7 days, which the first check only verifies that the difference is greater than 4 days. This means the second check is more strict than the first one, and the first one is therefore redundant.

Impact

Gas savings.

Recommendation

Remove the line `cliffAndFlowDate + START_DATE_VALID_AFTER >= endDate - END_DATE_VALID_BEFORE` because it is redundant with existing constant values.

Developer Response

No change necessary.

6. Gas - Remove unnecessary logic checks in `updateVestingSchedule()`

There are two logic checks in `updateVestingSchedule()` that are unnecessary and can be removed to save gas.

Technical Details

There are four if statements in `updateVestingSchedule()`. Some of them make sense and some don’t. Specifically the first two checks are unnecessary:

1.
   ```
   if (receiver == address(0) || receiver == sender) revert AccountInvalid();
   ```

   This check is not needed because no vesting can be scheduled with these function arguments. It is the same check that is found in `createVestingSchedule()`.

2.
   ```
   if (address(superToken) == address(0)) revert ZeroAddress();
   ```

   This check follows the same logic as the previous one. It is found in `createVestingSchedule()`.

If `updateVestingSchedule()` is called with an invalid combination of sender, receiver, and superToken, no vestingSchedule will be found in the mapping. That will mean `schedule.endDate == 0` and the call will revert on this line.

Impact

Gas savings.

Recommendation

Remove unnecessary logic to save gas.

Developer Response

Fixed in PR 77.

Informational Findings

1. Informational - Perform state changes earlier to mitigate possible reentrancy risk

The `executeCreateFlow()` function does not follow the checks-effects-interactions best practice for mitigating reentrancy. While the full external call flow was not fully analyzed due to the limited scope of this review, it is recommended to mitigate reentrancy risks by following the checks-effects-interactions pattern whenever possible.

Technical Details

The state variable changes in `executeCreateFlow()` are performed at the end of the function. An external call to ConstantFlowAgreementV1 is performed immediately before the state variables are modified. This is in contrast to `executeCliffAndFlow()` in VestingScheduler, where the state variable deletion happens before the external call, and this line even has a comment indicating the state variable change is performed early in the function for reentrancy mitigation.

The same pattern is found in `executeDeleteFlow()`. The delete operation happens after the external call, but it is likely better if the delete happens first and the external call happens at the end of the function.

Impact

Informational.

Recommendation

Follow the checks-effects-interactions best practices pattern for reentrancy mitigation. Move the `cfaV1.createFlowByOperator()` call to after the `flowSchedules` mapping deletions are performed.

Developer Response

Fixed in PR 77.

2. Informational - Deprecated app registration function called

`registerApp()` is deprecated but is still used as a backup to `registerAppWithKey()` in FlowScheduler and VestingScheduler.

Technical Details

`registerApp()` is marked as deprecated in ISuperfluid.sol. The Superfluid team controls the deprecation process for their own function, so the only case where unexpected failure modes could exist with the deprecation process is if this instance of `registerApp()` is forgotten in the future.

Impact

Informational.

Recommendation

Use a different alternative app registration process such as `registerAppByFactory()`.

Developer Response

Fixed in PR 77.

3. Informational - Inconsistencies between contracts

The FlowScheduler and VestingScheduler contracts are similar and have many lines of nearly identical code. It would be simpler if these two contracts maintained as much similarity as possible

1. for any Superfluid devs trying to understand these two contracts that did not originally write the contracts
2. to increase ease of comprehension in future code reviews.

Technical Details

One simple difference between the contracts that could be made more consistent is the declaration of state variables at the start of the contracts.

1. FlowScheduler state variables
2. VestingScheduler state variables

Another difference is whether the mapping deletion is done before or after the external call

1. FlowScheduler delete
2. VestingScheduler delete

Above are two examples but there are many others in the code. These include:

1. @dev NatSpec comments appears in VestingScheduler but not FlowScheduler
2. A startMaxDelay function argument in FlowScheduler that is not found in VestingScheduler, because VestingScheduler uses hard-coded values for `START_DATE_VALID_AFTER` and other constants
3. An assert in VestingScheduler that is absent from FlowScheduler
4. A `sender == address(0)` check in FlowScheduler that is missing from VestingScheduler
5. A `flowRate <= 0` check in VestingScheduler that is missing from FlowScheduler
6. The custom ScheduleAlreadyExists error in VestingScheduler but not FlowScheduler
7. A `flowDelayCompensation` compensation amount at the time of vesting execution in VestingScheduler that is not present in FlowScheduler
8. No transfer of an `earlyEndCompensation` amount when `executeDeleteFlow()` is called even though VestingScheduler's `executeEndVesting()` does this
9. Inconsistent naming between `executeDeleteFlow()` and `executeEndVesting()`
10. Different values of the mapping are deleted in `executeCliffAndFlow()` (only `cliffAndFlowDate` and `cliffAmount` are deleted) versus `executeCreateFlow()` (where the entire mapping is deleted in one case and `startDate`, `startMaxDelay`, and `flowRate` are deleted in the other case)
11. Varying failure modes for stream handling logic if the sender has insufficient tokens. In most cases the function call will revert, but in `executeEndVesting()` there is a try-catch clause to allow the function call to complete without reverting.
12. VestingScheduler does not include a `userData` value anywhere even though FlowScheduler does. This `userData` value is passed into the `cfaV1.createFlowByOperator()` call when called in `executeCliffAndFlow()`. The

`userData` value is used in setting the ctx in the host Superfluid.sol `callAgreement()` function. The `cfaV1.deleteFlowByOperator()` call also has this `userData` difference between the two contracts.

13 The state machine diagrams demonstrate that FlowScheduler has an extra ability to call `createFlowSchedule()` to update (by overwriting) a scheduled flow immediately after the flow is scheduled. The flow mapping can also be overwritten after the flow has started. VestingScheduler does not allow this because of a revert check that prevents overwriting of an existing mapping. It is not clear whether these should be a difference between the two contracts here, and if there should not be a difference, there are pros and cons to either approach.

It would be best for the devs to perform their own comparison of similar contracts prior to launch to standardize the logic and catch other discrepancies not mentioned here, because this is not a comprehensive list.

Impact

Informational.

Recommendation

Keep the code in similar contracts as similar as possible. One way to do this is to abstract the logic shared between the two Scheduler contracts to a parent contract, let’s say ParentScheduler. This ParentScheduler contract can contain shared logic like `_getSender()` and abstract versions of functions like `deleteFlowSchedule()` or `deleteVestingSchedule()` if the logic is similar enough in places to abstract it to an internal function in an inherited parent contract.

Developer Response

Relevant observations are fixed in PR 77. Many design differences are intentional.

4. Informational - Inconsistent failure modes

The try-catch clause in `executeEndVesting()` allows the stream to end even if the sender does not have enough funds to cover the `earlyEndCompensation` transfer. All other cases where the sender has insufficient funds will cause the function call to revert. This inconsistency could lead to user confusion and poor decision making when interacting with the protocol.

Technical Details

Normally if a user does not have sufficient funds for a `transfer` or `transferFrom` to complete, the code will revert. There is one `transferFrom()` in FlowScheduler and two in VestingScheduler. Similarly, if a Superfluid stream is started but the sender does not have sufficient tokens to meet the minimum buffer requirements, it should revert. There is a call to start a Superfluid stream in FlowScheduler and in VestingScheduler. The final `transferFrom` in VestingScheduler is in a try-catch clause and it is the only instance of this type that will not revert if the sender has insufficient funds. Whether it is correct for this try-catch to appear in only one place or whether this edge case is inconsistent with planned failure modes is up to the protocol developers. But this try-catch will cause the receiver to receive less tokens than expected in a way that other `transferFrom` calls in these contracts do not.

Impact

Informational. End result may or may not match developer intent.

Recommendation

If the code remains the same, clearly document the reason why a try-catch in this one location makes sense and is not applied to all `transferFrom` calls.

Developer Response

It doesn’t make sense to revert the closing of the stream. If the stream couldn’t be closed then it eventually would be closed by liquidation.

5. Informational - Mappings may not be cleared after stream ended

If a sender or receiver decides to end the Superfluid stream without using the Scheduler contract `executeDeleteFlow()` or `executeEndVesting()`, but instead follow a different path to ending the stream, the mapping that stores this scheduled flow or vesting will not be cleared out completely due to a revert condition. While this is not likely to cause any security issues, the contract may end up storing a large number of useless values in the mapping, which could hurt gas efficiency.

Technical Details

After a scheduled stream has started, the sender and receiver do not necessarily need to use the Scheduler contracts. Instead, they can control the stream started by the Scheduler contracts in the same way as a normal Superfluid stream. If this stream is ended early, or even ended on time but without the `executeDeleteFlow()` or `executeEndVesting()` functions getting called, then it will not be possible to call `executeDeleteFlow()` or `executeEndVesting()` because the `cfaV1.deleteFlowByOperator()` call will revert when the stream no longer exists. In the case of FlowScheduler, this can mean that the `endDate` and `userData` remain in the mapping until it is overwritten by another scheduled stream or until the sender calls `deleteFlowSchedule()` to delete the remaining fields of the mapping. For VestingScheduler, the `endDate` and `flowRate` values may remain in the mapping. There do not appear to be any code paths that would create problems when values remain in the mapping, but the lack of a guarantee that the mapping has been fully cleared is not ideal.

Impact

Informational.

Recommendation

Allow `deleteFlowSchedule()` and `deleteVestingSchedule()` (or similar functions) to be called by anyone for any mapping that contains an expired `endDate` value. Alternatively, consider placing the `cfaV1.createFlowByOperator()` call in a try-catch statement unless potentially skipping this external call has side effects.

Developer Response

Fixed in PR 77.

6. Informational - Unused named return variables

Several functions such as `executeCreateFlow()`, `executeDeleteFlow()`, `executeCliffAndFlow()`, and `executeEndVesting()` declared named return variables but don’t use them.

Technical Details

These return variables do not need to be named, or should be set to `true` at the end of the function instead of returning `true` directly.

1. FlowScheduler state variables
2. VestingScheduler state variables

Impact

Informational.

Recommendation

Remove named return variables or set these variables to the proper return value at the end of each function.

Developer Response

Returning the variable name in this case would be redundant and not as direct as the current approach.

7. Informational - Unclear flow and vesting update process

The existing NatSpec does not make it very clear that the FlowScheduler `createFlowSchedule()` is intended to be used for the update process by overwriting existing values. When compared to VestingScheduler, it looks like FlowScheduler is missing an update function. Additionally, it is unclear that the intended path for updating a scheduled vesting that needs multiple arguments changed (not just changing `endDate`) is to delete the vesting and create a new vesting with the proper arguments.

Technical Details

VestingScheduler has a `updateVestingSchedule()` function that allows existing vestings to be updated. FlowScheduler does not have a similar update function, so it is not obvious that `createFlowSchedule()` is designed to allow overwriting of existing flows. Clarifying this in documentation will make it easier for developers updating Superfluid in the future, developers integrating with Superfluid features, and future security analysis of the contracts.

Impact

Informational.

Recommendation

Document that FlowScheduler’s `createFlowSchedule()` serves two purposes: creating new flows and updating existing flows. It may be best for FlowScheduler and VestingScheduler to use the same update path for consistency: reusing the create function to overwrite existing entries in the mapping.

Developer Response

Fixed in PR 77.

8. Informational - Some comments don’t match the code

Some comments don’t match the code or poorly describe the code.

Technical Details

Several comments are incorrect or do not match the current contract code:

1. This line of FlowScheduler indicates the FlowSchedule struct only has 4 fields. The actual definition in IFlowScheduler.sol has 6 fields.
2. This line of FlowScheduler should read `ID = KECCAK(superToken, sender, receiver)` and not `ID = KECCAK(sender, receiver, superToken)`.
3. The IVestingScheduler NatSpec indicates that an `endDate` with a value of zero is an acceptable scenario (here and here). But there is a foundry test in VestingScheduler.t.sol that specifically checks that an `endDate` of zero will revert when scheduling a future vesting.
4. To reduce the confusion of using “flowing” and “vesting” in the same sentence in the vesting contract, this comment in VestingScheduler might be better written as “Only allow an update if 1. vesting exists 2. executeCliffAndFlow() has not been called”

Impact

Informational.

Recommendation

Update comments to match the latest code.

Developer Response

Fixed in PR 77.

9. Informational - Documentation wiki has many TODOs

The Superfluid wiki at https://github.com/superfluid-finance/protocol-monorepo/wiki contains documentation about the system. It is incomplete and has many missing paragraphs or pages. This should be fixed to make sure developers use and interface with the protocol properly.

Technical Details

The wiki is missing many pages. For example, all pages in the Governance Overview do not exist, and even this summary page contains some TODO comments.

A tangential comment on the main docs page: the words “as of Q4 2021” on this page should be updated to “as of Q4 2022”.

Impact

Informational.

Recommendation

If the wiki is no longer used, remove it. If it is still used, fix all TODOs to avoid developers incorrectly integrating with Superfluid.

Developer Response

Acknowledged, though this is not relevant to the Scheduler.

10. Informational - Non-zero allowance to Superfluid contract could lead to phishing

Non-zero allowances, especially infinite allowances, are generally a concern for users because it implies trusting other addresses with their tokens. No concrete attack vectors were found to leverage such an allowance, but because Super Apps use a custom `flowRateAllowance` allowance system, enable custom code, and implement callbacks, phishing attacks targeting Superfluid users may use this attack vector.

Technical Details

Users should not interact with malicious contracts. However, it is impossible to guarantee user safety when the issue is between the chair and the keyboard, so monitoring for unexpected sudden outflows from addresses that have non-zero `flowRateAllowance` allowances is an important mechanism for monitoring potential attack vectors targeting non-zero allowances. This concern is not unique to Superfluid, but Superfluid does add some unique complexities around this vector.

Impact

Informational.

Recommendation

It is advisable for Superfluid to monitor addresses that have non-zero `flowRateAllowance` allowances set in order to be alerted of any large volume flows that could indicate malicious activity. The Superfluid frontend should limit the default approval allowance set for users and ideally suggest users to reset their approval to Superfluid contracts back to zero after the stream ends. This may not be ideal from a user experience perspective, but it does improve the user's security. One approach to returning approvals to zero is to allow a keeper to set the approval to zero if the address has not started a

Superfluid stream after a set time period (potentially a customizable time period value that users can select based on their security preferences).

Developer Response

No action needed, this works like `safeApprove` for ERC20 tokens.

Closing Thoughts

I see a few overarching areas where additional focus could increase the clarity of how the protocol is supposed to function and result in cleaner code with fewer unexpected edge cases.

1. Map out the expected flow of a scheduled flow or vesting. A tool like Stately can help with this. By drawing out the allowed function call paths, it becomes easier to see which paths are not supposed to happen. Logical checks can be added to make sure the calls happen only in this order. Proceeding in this more rigorous manner would reduce unforeseen cases where functions are called out of order, resolving some issues found in this report in the process.
2. Make a plan to improve the alignment between the similar contracts FlowScheduler and VestingScheduler. This plan could take many forms. One option is to keep the contracts as they are but add newlines to keep the line numbers more aligned so that equivalent lines of code appear on the same line number in the two contracts. Another idea is to abstract shared logic to a parent contract to avoid rewriting similar code, and allowing changes to happen in only one place instead of two. Yet another option is to periodically review or document differences in the two contracts to make sure that all differences are intentional and serve a clear purpose. If no alignment strategy is in place, an improvement may be added to one contract but not the other, and the combination of many such differences could result in oversights in security checks or similar problematic scenarios.
3. When describing the scheduling contracts, the development team commented several times that the sender or receiver have ultimate control over the stream and can cancel it at any time. That may be true, but such assumptions or expectations of user behavior should be clearly documented and enablers for a “best case” situation be put into place. One example of this is that there are two ways to update a future

scheduled vesting: call `updateVestingSchedule()` to update the `endDate`, or call `deleteVestingSchedule()` and then `createVestingSchedule()` to create a new vesting that can update any field (not just `endDate`). The question is which of these two options is best for the user, and should two options to accomplish the same task exist? Similarly, VestingScheduler has a `flowDelayCompensation` and `earlyEndCompensation` calculation to provide the receiver with a perfectly calculated amount of tokens for the duration the stream was scheduled to run for. Is there a reason that the users of FlowScheduler should not have the same level of precision? Another example is that because of this logic branch in `executeCreateFlow()`, it should be documented that `executeDeleteFlow()` is not intended to ever be called if `endDate == 0` because the mapping in `flowSchedules` would already be fully deleted in that case. A good start at documenting these assumptions is found in this Vesting Scheduler doc, but this document doesn't accurately describe the Flow Scheduler and leaves out some important assumptions like those mentioned previously.
