CERTORA

Formal Verification Report of Radicle Drips Contracts

Summary

This document describes the specification and verification of **Radicle Drips Contracts** using the Certora Prover. The work was undertaken from **06 December 2022** to **01 January 2023**. The latest commit that was reviewed and run through the Certora Prover was **6fe9d38**.

The scope of our verification was the Splits contract `Splits.sol`.

The Certora Prover proved the implementation of the contract above is correct with respect to the formal rules written by the Certora team. The team also performed a manual audit of the contract.

All the rules are publically available and can be found in **Radicle's public github**.

Main Issues Discovered

Informational

Issue:	**A split receiver might be prevented from receiving tokens**
Description:	A split receiver's part can be rounded down to zero, if `split()` is called upon a splitter, when the `splittable amount of the splitter x split receiver's weight / total splits weight` is rounded down to zero. The one who will benefit from the above is the splitter who will get the remainder to himself. It is possible for a splitter to abuse the above, if he (or a pre-designed bot) calls `split()` constantly without allowing for a substantial splittable amount to be gathered and then split.
Response:	- The splitting user can't choose how much they want to split. The funds come from drips, splits and giving, all of which are barely controllable and certainly not enough to pinch token after token. - The splitting user's configuration is based on their good will anyway. They can choose to update their configuration before splitting and just take all the pending funds, not just a single token. The only risk is that they can be frontrun by a bot splitting with their previous configuration, and of course

	reputation, which is the main penalty here. When you drip, split or give to somebody, you're basically letting them control the funds however they feel fit. - There are very few tokens for which pinching tokens like this justifies the gas expense required to perform an attack. - There's really no good workaround, splitting 1 undividable token between 2 parties must leave one of them with a token and one without. An alternative could be to either lock the disputed token, or keep track of who gained more, but that'd be very complex and almost certainly unreliable.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful but we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

- We unroll loops. Violations that require a loop to execute more than twice will not be detected
- In some of the rules we assume that the weights of the split receivers are assigned correctly, i.e. sum of all weights is less than `_TOTAL_SPLITS_WEIGHT`. This is a safe assumption since the function `_assertSplitsValid()` verifies that in `_setSplits()`
- Since the rule *cannotFrontRunSplitGeneral* timed-out in the general case with no assumptions on userA, userB and userC, we had to split the rule into different simpler rules with specific assumptions for the users and then prove each rule on its own
- We assumed that the sum of balances and the amounts transferred are less than 2^{128}

Notations

✅Indicates the rule is formally verified.
❌Indicates the rule is violated.
⏳Indicates the rule is timing out.

Verification of Splits.sol

Properties and Rules

✅ correctnessOfSplitResults

Calculate results of splitting an amount using the current splits configuration.
We verify that `amount == collectableAmt + splitAmt`

✅ correctnessOfSplit

Splits user's received but not split yet funds among receivers.
We verify that for the splitter:
`splittableBefore >= splittableAfter`
`collectableBefore + collectableAmt == collectableAfter`
`splittableBefore + collectableBefore >= splittableAfter + collectableAfter`

✅ integrityOfCollect

Collects user's received already split funds
We verify that after collection, no more collectable balance should be immediately available
`collectableAfter == 0`

✅ revertCharacteristicsOfCollect

We verify that calling the method `collect()` should never revert

✅ correctnessOfGive

The method `give()` gives `amt` amount of `assetId` funds from `userId` to `receiver`.
We verify that after giving `amt` to a receiver, the splittable of the receiver should increase exactly by `amt`.

✅ splittableOfNonReceiverNotAffectedByGive

The method `give()` gives `amt` amount of `assetId` funds from `userId` to `receiver`.
We verify that the splittable of any other user that is not receiver should not change

✅ correctnessOfHashSplits

The method `hashSplits()` calculates the hash of the list of splits receivers.
We use a boolean selector to operate on two different lists of splits receivers.
We verify that two calculated hashes are the same only if they got exactly the same input.
We also verify that different inputs must generate different hashes.

✅ integrityOfSplit

We simulate the following scenario:
userA has received drips and has splittable > 0
userA has a list of splitters that should get some of the drips received
userA calls `split()`

Then we verify the following:
If userB is on the list of userA's splitters, therefore userB's splittable should NOT decrease
If userC is NOT on the list of userA's splitters, therefore userC's splittable should NOT change
userA's collectable should NOT decrease
userB and userC's collectable should NOT change

✅ assetsDoNotInterfereEachOther

We verify that operations over one `assetId1` should not affect anything related to another `assetId2`
We verify that calling `split()` on userA with `assetId1` should NOT affect the splittable and collectable for any user's `assetId2`

✅ moneyNotLostOrCreatedDuringSplit

Money is not lost or created in the system when split() is called.
We simulate the following scenario: userA has splittable balance and one splits receiver - userB, then `split()` is called on userA.
We verify that the sum (splittable + collectable) of (userA + userB) are invariant of the split.

✅ sameReturnOfSplitAndSplitResults

We verify that `splitResults()` and `split()` return the same `(collectableAmt, splitAmt)`
We simulate the following scenario:
userA has splittable balance and configured two split receivers - userB, userC
First we call `splitResults()` upon userA with `amount = splittable of userA`, then we call `split()` upon userA
We verify that the returned values of both functions are the same

❌ splitReceiverShouldGetMoneyUponSplit

We simulate the following scenario:
userA has splittable balance and one splits receiver - userB, then `split()` is called on userA
We verify that the splittable balance of userB will increase

The rule fails in cases when userB's part, calculated as (`splittable amount of userA x userB's weight / _TOTAL_SPLITS_WEIGHT`) is rounded down to zero, therefore the receiver will get nothing!

Possible abuse vector: `split()` is called every time when the splittable balance of userA is so low, so that the rounding error will cause the splitReceiver userB to get zero. As a result userA will get all the splittable to himself.

The one who will benefit from the abuse is the splitter, but he is also the one that in advance decided who are going to be his splitReceivers.

✅ equalSplitWeightsResultEqualSplittableIncrease

Users with same weights should get same amount upon `split()`
We simulate the following scenario: userA has splittable balance and configured two split receivers - userB, userC. Both userB and userC have the same split weights, then `split()` is called on userA.
We verify that the splittable balances of userB and userC will increase by the same amount (up to 1 unit)

⏳ cannotFrontRunSplitGeneralCase

Front running `split()` does not affect the split receiver
We simulate the following scenario:
userA has a single splitReceiver userC
userB also has the same single splitReceiver UserC
we want to verify that `split()` can be called on userA successfully even if someone front runs it and calls `split()` first on userB

This rule has no assumptions about userA, userB, userC and unfortunately the rule timed out. In order to deal with the time out, we decided to verify the rule by dividing it into the following simpler rules with explicit assumptions about the identity of the users. All the simpler rules were verified successfully.

✅ cannotFrontRunSplitDifferentUsers

First we verify the case `userA != userB != userC`

✅ cannotFrontRunSplitTwoSameUsers

Next we verify the case `userA != userB` with appropriate require

✅ cannotFrontRunSplitThreeSameUsers

Finally we verify the edge case `userA == userB == userC`