Drips Protocol contest Findings & Analysis Report

2023-03-10

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Drips Protocol smart contract system written in Solidity. The audit contest took place between January 25—February 3 2023.

Wardens

26 Wardens contributed reports to the Drips Protocol contest:

1. 0xA5DF
2. 0xSmartContract
3. 0xbepresent
4. Aymen0909
5. Deivitto
6. HollaDieWaldfee
7. IllIIII
8. NoamYakov
9. ReyAdmirado
10. Rolezn
11. SleepingBugs (Deivitto and 0xLovesleep)
12. berndartmueller
13. btk
14. chaduke
15. cryptostellar5
16. descharre
17. evan
18. fs0c
19. hansfriese
20. hihen

21. ladboy233
22. matrix_0wl
23. nalus
24. rbserver
25. zzzitron

This contest was judged by Alex the Entreprenerd.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 3 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 2 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 15 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 11 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Drips Protocol contest repository, and is composed of 8 smart contracts written in the Solidity programming language and includes 1,503 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges

- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](), specifically our section on [Severity Categorization]().

High Risk Findings (1)

[H-01] Drips that end after the current cycle but before its creation can allow users to profit from squeezing

Submitted by [evan](), *also found by* [HollaDieWaldfee]()

By creating a drip that ends after the current cycle but before its creation time and immediately removing it, the sender doesn't have to put in any assets but the receiver can still squeeze this drip.

By setting a receiver that the sender controls, the sender can drain an arbitrary asset from the contract.

Proof of Concept

Let the cycle length be 10 seconds. By i-th second I mean the i-th second of the cycle.

At the 5th second, sender creates a drip that starts at 0th second and lasts for 2 seconds.

At the 6th second, sender removes this drip.

https://github.com/code-423n4/2023-01-drips/blob/main/src/Drips.sol#L569

Since the drip ends before it was created, the dripped amount is 0, so the sender can retrieve their full balance.

https://github.com/code-423n4/2023-01-drips/blob/main/src/Drips.sol#L425-L430 https://github.com/code-423n4/2023-01-drips/blob/main/src/Drips.sol#L490-L496

Now the receiver squeezes from this drip.

`SqueezeStartCap = \_currCycleStart() = 0th second, squeezeEndCap = 6th second`, so the receiver can still squeeze out the full amount even though the sender has withdrawn all of his balance.

Please add the following test to DripsHub.t.sol. It verifies that the sender has retrieved all of his assets but the receiver can still squeeze.

```
    function customSetDrips(
        uint256 forUser,
        uint128 balanceFrom,
        uint128 balanceTo,
        DripsReceiver[] memory newReceivers
    ) internal {
        int128 balanceDelta = int128(balanceTo) - int128(balance
        DripsReceiver[] memory currReceivers = loadDrips(forUser

        vm.prank(driver);
        int128 realBalanceDelta =
            dripsHub.setDrips(forUser, erc20, currReceivers, bal

        storeDrips(forUser, newReceivers);

    }

    function testExploitSqueeze() public {
        skipToCycleEnd();
        // Start dripping
        DripsReceiver[] memory receivers = new DripsReceiver[](1
        receivers[0] = DripsReceiver(
            receiver,
            DripsConfigImpl.create(0, uint160(1 * dripsHub.AMT_P
        );

        DripsHistory[] memory history = new DripsHistory[](2);

        uint256 balanceBefore = balance();
        skip(5);
        customSetDrips(user, 0, 2, receivers);
        (,, uint32 lastUpdate,, uint32 maxEnd) = dripsHub.dripsS
        history[0] = DripsHistory(0, receivers, lastUpdate, maxE

        skip(1);
        receivers = dripsReceivers();
        customSetDrips(user, 2, 0, receivers);
        (,, lastUpdate,, maxEnd) = dripsHub.dripsState(user, erc
        history[1] = DripsHistory(0, receivers, lastUpdate, maxE

        assertBalance(balanceBefore);

        // Squeeze
        vm.prank(driver);
        uint128 amt = dripsHub.squeezeDrips(receiver, erc20, use
        assertEq(amt, 2, "Invalid squeezed amt");
    }
```

Tools Used

VS Code, Foundry

Recommended Mitigation Steps

https://github.com/code-423n4/2023-01-drips/blob/main/src/Drips.sol#L426

One potential solution is to add an additional check after this line. Something along the lines of:

```
 if (squeezeStartCap < drips.updateTime) squeezeStartCap =
drips.updateTime;
```

CodeSandwich (Drips) confirmed and commented:

> Great job! This is a critical protocol breaker.

Alex the Entreprenerd (judge) commented:

> The Warden has shown a way to trick the contract into disbursing out funds without the upfront payment.

> Because this shows a way to steal the principal, I agree with High Severity.

Medium Risk Findings (2)

[M-01] Squeezing drips from a sender can be front-run and prevented by the sender

Submitted by berndartmueller

https://github.com/code-423n4/2023-01-drips/blob/9fd776b50f4be23ca038b1d0426e63a69c7a511d/src/Drips.sol#L411

https://github.com/code-423n4/2023-01-drips/blob/9fd776b50f4be23ca038b1d0426e63a69c7a511d/src/Drips.sol#L461

Squeezing drips from a sender requires providing the sequence of drips configurations (see NatSpec description in L337-L338):

> /// It can start at an arbitrary past configuration, but must describe all the configurations /// which have been used since then including the current one, in the chronological order.

The provided history entries are hashed and verified against the sender's `dripsHistoryHash`.

However, the sender can prevent a receiver from squeezing drips by front-running the squeeze transaction and adding a new configuration. Adding a new configuration updates the current `dripsHistoryHash` and invalidates the `historyHash` provided by the receiver when squeezing. The receiver will then fail the drips history verification in L461 and the squeeze will fail.

Impact

A sender can prevent its drip receivers from squeezing by front-running the squeeze transaction and adding a new configuration.

Proof of Concept

Drips.sol#L411

```
function _squeezeDripsResult(
    uint256 userId,
    uint256 assetId,
    uint256 senderId,
    bytes32 historyHash,
    DripsHistory[] memory dripsHistory
)
    internal
    view
    returns (
        uint128 amt,
        uint256 squeezedNum,
        uint256[] memory squeezedRevIdxs,
        bytes32[] memory historyHashes,
        uint256 currCycleConfigs
    )
{
    {
        DripsState storage sender = _dripsStorage().states[
        historyHashes = _verifyDripsHistory(historyHash, dr
        // If the last update was not in the current cycle,
        // there's only the single latest history entry to
        currCycleConfigs = 1;
        // slither-disable-next-line timestamp
        if (sender.updateTime >= _currCycleStart()) currCyc
    }
    // [...]
```

Drips.sol#L461

```
function _verifyDripsHistory(
    bytes32 historyHash,
    DripsHistory[] memory dripsHistory,
    bytes32 finalHistoryHash
) private pure returns (bytes32[] memory historyHashes) {
    historyHashes = new bytes32[](dripsHistory.length);
    for (uint256 i = 0; i < dripsHistory.length; i++) {
        DripsHistory memory drips = dripsHistory[i];
        bytes32 dripsHash = drips.dripsHash;
        if (drips.receivers.length != 0) {
            require(dripsHash == 0, "Drips history entry wi
            dripsHash = _hashDrips(drips.receivers);
        }
        historyHashes[i] = historyHash;
```

```
            historyHash = _hashDripsHistory(historyHash, dripsH
        }
        // slither-disable-next-line incorrect-equality,timesta
        require(historyHash == finalHistoryHash, "Invalid drips
    }
```

Recommended Mitigation Steps

Consider allowing a receiver to squeeze drips from a sender up until the current timestamp.

[xmxanuel (Drips) commented]():

> Technically the described attack would work.

> It is also related to the trust assumption between the sender an receiver. Which are anyway given in a certain form.

> I am unsure about the proposed solutions. Currently, we only store the latest dripsHistory hash on-chain. For allowing to squeeze until a specific timestamp, it might be necessary to have the full list of historic hashes on-chain.

[CodeSandwich (Drips) disagreed with severity and commented]():

> This attack is in fact possible, but it only allows postponing collecting funds until the end of the cycle. The proposed solution would probably require a lot of storage, so it's probably not worth introducing.

[Alex the Entreprenerd (judge) commented]():

> In order to judge this issue I spoke with 4 other judges as well as the Sponsor.

> This is a difficult decision because of the unclear expectations as to whether the sender is bening towards the receiver or not.

> Because of the uncertainty, and the possibility of performing the grief, the finding is valid.

> The issue is in terms of determining it's severity.

> **The attacker doesn't gain anything**

> One of the most interesting arguments is the idea that "the attacker doesn't gain anything", which can be falsified in scenarios in which squeezing would have helped reach a threshold of tokens in circulation, out of vesting or for similar purposes.

> If we try hard enough, through multiple people's effort, we can come up with a scenario in which squeezing could make a difference between having enough votes for a Governance Operation, or could offer enough collateral to avoid a liquidation or some level of risk.

> Those scenarios, while unlikely, can help discredit the idea that "the attacker doesn't gain anything".

> **The grief is limited**

> On the other hand, we must acknowledge that the attack / griefing, is limited:

- The `sender` is the only privileged account that can perform the grief
- Squeezing is denied until the next cycle, which based on configuration, will be known, and will be between 1 week and 1 month.

> These can be viewed as additional external requirements, which reduce the impact / likelihood of the finding

> **The Squeeze is Squoze**

> In spite of the external requirements, the finding is showing how the functionality of Squeezing can be denied by the sender.

> I believe that if anybody could grief squeezing, we would not hesitate in awarding Medium Severity and perhaps we'd be discussing around Med / High.

> However, in this case, the only account that can perform the grief is the sender.

> At the same time, the goal of the system is to allow Squeezing, which per the discussion above, given the finding can be prevented until a drip has moved to a newer crycle.

Last Minute Coding

I went into the tests and wrote the following illustratory cases

```
  function testSenderCanStopAfter() public {
        uint128 amt = cycleSecs;
        setDrips(sender, 0, amt, recv(receiver, 1), cycleSecs);
        DripsHistory[] memory history = hist(sender);
        skip(1);
        squeezeDrips(receiver, sender, history, 1);
        setDrips(sender, amt - 1, 0, recv(receiver, 1), 0);

        // Squeeze again
        DripsHistory[] memory history2 = hist(history, sender);
        squeezeDrips(receiver, sender, history2, 0);

        skipToCycleEnd();
        receiveDrips(receiver, 0);
    }

    function testSenderCannotChangeTheirMind() public {
        uint128 amt = cycleSecs;
        setDrips(sender, 0, amt, recv(receiver, 1), cycleSecs);
        DripsHistory[] memory history = hist(sender);
        skip(1);
        setDrips(sender, amt - 1, 0, recv(receiver, 1), 0);

        // Squeeze First time
        DripsHistory[] memory history2 = hist(history, sender);
        squeezeDrips(receiver, sender, history2, 1);

        skipToCycleEnd();
        receiveDrips(receiver, 0);
    }
```

Ultimately we can see that what is owed to the recipient can never be taken back, however the sender can, through the grief shown in the above finding, prevent the tokens from being received until the end.

Conclusion

Given the information above, being mindful of:

- A potentially severe risk, with very low likelyhood
- The temporary breaking of the functionality of squeezing, which was meant to allow by-the-second claims.

> I agree with Medium Severity.

[M-02] `unauthorize()` can be front-run so that the malicious authorized user would get their authority back

Submitted by [0xA5DF](), also found by [0xbepresent]() and [ladboy233]()

The `Caller` contract enables users to authorize other users to execute tx on their behalf. This option enables the authorized/delegated user to add more users to the authorized users list. In case the original user is trying to remove an authorized user (i.e. run `unauthorize()`), the delegated user can simply front run that tx to add another user, then after `unauthorized()` is executed the delegated can use the added user to gain his authority back.

This would allow the malicious delegated user to keep executing txs on behalf of the original user and cause them a loss of funds (e.g. collecting funds on their behalf and sending it to the attacker's address).

Proof of Concept

The test below demonstrates such a scenario:

- Alice authorizes Bob
- Bob becomes malicious and Alice now wants to remove him
- Bob noticed the tx to unauthorize him and front runs it by authorizing Eve
- Alice `unauthorize()` tx is executed
- Bob now authorizes himself back again via Eve's account

Front running can be done either by sending a tx with a higher gas price (usually tx are ordered in a block by the gas price / total fee), or by paying an additional fee to the validator if they manage to run their tx without reverting (i.e. by sending additional ETH to `block.coinbase`, hoping validator will notice it). It's true that Alice can run `unauthorize()` again and again and needs to succeed only once, but:

- Bob can keep adding other users and Alice would have to keep removing them all
- This is an ongoing battle that can last forever, and Alice might not have enough knowledge, resources and time to deal with it right away. This might take hours or days, and in the meanwhile Alice might be receiving a significant amount of drips that would be stolen by Bob.

```
diff --git a/test/Caller.t.sol b/test/Caller.t.sol
index 861b351..3e4be22 100644
--- a/test/Caller.t.sol
+++ b/test/Caller.t.sol
@@ -125,6 +125,24 @@ contract CallerTest is Test {
         vm.expectRevert(ERROR_UNAUTHORIZED);
         caller.callAs(sender, address(target), data);
     }
+
+    function testFrontRunUnauthorize() public {
+        bytes memory data = abi.encodeWithSelector(target.run.s
+        address bob = address(0xbab);
+        address eve = address(0xefe);
+        // Bob is authorized by Alice
+        authorize(sender, bob);
+
+        // Bob became malicious and Alice now wants to remove h
+        // Bob sees the `unauthorize()` call and front runs it 
+        authorizeCallAs(sender, bob, eve);
+
+        unauthorize(sender, bob);
+
+        // Eve can now give Bob his authority back
+        authorizeCallAs(sender, eve, bob);
+
+    }

     function testAuthorizingAuthorizedReverts() public {
         authorize(sender, address(this));
@@ -257,6 +275,13 @@ contract CallerTest is Test {
         assertTrue(caller.isAuthorized(authorizing, authorized)
     }

+    function authorizeCallAs(address originalUser,address deleg
+        bytes memory data = abi.encodeWithSelector(Caller.autho
+        vm.prank(delegated);
+        caller.callAs(originalUser, address(caller),  data);
+        assertTrue(caller.isAuthorized(originalUser, authorized
+     }
+
     function unauthorize(address authorizing, address unauthori
         vm.prank(authorizing);
         caller.unauthorize(unauthorized);
```

Recommended Mitigation Steps

Don't allow authorized users to call `authorize()` on behalf of the original user.

This can be done by replacing `_msgSender()` with `msg.sender` at `authorize()` , if the devs want to enable authorizing by signing I think the best way would be to add a dedicated function for that (other ways to prevent calling `authorize()` from `callAs()` can increase gas cost for normal calls, esp. since we'll need to cover all edge cases of recursive calls with `callBatched()`).

xmxanuel (Drips) commented:

> Known behavior. I think it is debatable. The `authorize` means an other address is completely trusted in terms of access to funds. In most cases this will be the same actor like Alice. Just another address of Alice.

CodeSandwich (Drips) disagreed with severity and commented:

> Like Manuel said, this is known, authorizing somebody gives them extreme privileges anyway, like stealing all the funds available for the user in the protocol.

> I disagree that it's a High severity issue, because the attacked user with some effort can defend themselves. In a single batch they need to authorize a contract that would iterate over all addresses returned by `allAuthorized` and for each of them to call `unauthorize` on behalf of the attacked user.

> I think that the above emergency defense mechanism could be built right into `Caller`'s API to make it cheaper and easier to run, basically adding an `unauthorizeAll` function that will clear the authorized addresses list. That's why I agree that it **is** an issue to solve, but a low severity one, it's only a quality of life improvement.

> The proposed mitigation of banning authorized users from authorizing others is too heavy handed, because it hurts use cases like a slow DAO authorizing a trusted operator, who then needs to rotate keys or authorize a “script contract” performing automated operations on Drips protocol. Authorization by signing hurts all non-EOA addresses like DAOs or multisigs which don’t have private keys.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> Because of the fact that the only requirement for the attack is for the authorized to be malicious, I have considered High Severity.

> After further reflection, because of the ability to remove the malicious attacker via a Macro, per the Sponsor’s comment above, I believe Medium Severity to be the most appropriate.

> The approval should be considered an extremely dangerous operation, which can cause drastic losses, however, a malicious approved can eventually be unapproved via a macro contract.

xmxanuel commented:

> The approval should be considered an extremely dangerous operation, which can cause drastic losses, however, a malicious approval can eventually be unapproved via a macro contract

> Yes, we agree here. We envision it more like the same actor/organization is using multiple addresses.

> Like the multi-sig of an organization is used as the main address (address with the ens name) but not all actions might require multiple signatures. Therefore, they can approve another address. etc.

> However, we assume full trust.

Alex the Entreprenerd (judge) commented:

> Confirming Medium severity because:

- The risk is not in giving the approval, that is the feature offered

- The risk is in not being able to revoke an approval, due to a lack of `revokeAll` or similar functionality, which can create a temporary grief

> Because that can cause a denial of functionality which is limited in time, I believe Medium Severity to be the most appropriate.

Low Risk and Non-Critical Issues

For this contest, 13 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **berndartmueller** received the top score from the judge.

The following wardens also submitted reports: SleepingBugs, rbserver, hansfriese, IllIllI, 0xSmartContract, nalus, zzzitron, 0xA5DF, fs0c, chaduke, btk, and HollaDieWaldfee .

[L-01] An authorized user can unauthorize other authorized users of the same sender

A user can grant authorization to another address to make calls on their behalf via the `Caller.callAs` function. The `Caller.unauthorize` function allows the user to revoke the authorization of another address.

An authorized user can revoke the authorization of another authorized user of the same sender. This is because the authorized user can call the `Caller.unauthorize` function on behalf of the sender.

Findings

Caller.sol#L114-L118

```
114: function unauthorize(address user) public {
115:     address sender = _msgSender();
116:     require(_authorized[sender].remove(user), "Address is n
117:     emit Unauthorized(sender, user);
118: }
```

Recommended Mitigation Steps

Consider preventing calls to the `Caller` contract address from within the `_call` function.

[L-02] Lack of reasonable boundaries for cycle secs

If `_cycleSecs` is set too low, for example, to a value less than the Ethereum block time of 12 sec, it will not be possible to squeeze. If set too high, e.g., larger than the current `block.timestamp`, it will not be possible to receive drips - only squeezing will be possible.

Findings

[Drips.sol#L221]()

```
219: constructor(uint32 cycleSecs, bytes32 dripsStorageSlot) {
220:     require(cycleSecs > 1, "Cycle length too low");
221:     _cycleSecs = cycleSecs;
222:     _dripsStorageSlot = dripsStorageSlot;
223: }
```

Recommended Mitigation Steps

Consider adding an appropriate upper limit for `_cycleSecs`.

[L-03] Unused `dripId` of receiver config is used considered while sorting receivers

The `Drips._isOrdered` function compares two given receivers. The receiver config can include the prefixed `dripId` in the 32 most significant bits. This allows using the `dripId` for changing the order of receivers when setting a new drips configuration without violating the sorting requirement, *First compares their* `amtPerSec` *s, then their* `start` *s and then their* `duration` *s* ([see Drips.sol#L93]()).

While no harmful effects are expected, it can be used in the `Drips._setDrips` function to control if delta amounts of receivers are first removed, added, or updated, are expected.

Findings

[Drips.sol#L1069]

```
1061: function _isOrdered(DripsReceiver memory prev, DripsReceiv
1062:     private
1063:     pure
1064:     returns (bool)
1065: {
1066:     if (prev.userId != next.userId) {
1067:         return prev.userId < next.userId;
1068:     }
1069:     return prev.config.lt(next.config);
1070: }
```

Recommended Mitigation Steps

Consider omitting the `dripId` from the receiver config when comparing the receivers.

[L-04] Collecting funds should be usable while the `DripsHub` contract is paused

The `DripsHub.collect` function is only callable when the `DripsHub` contract is not paused. This prevents a user from collecting accumulated funds. The admin of the `DripsHub` contract can potentially pause the contracts at any time, locking users out of their honestly earned funds.

Findings

[DripsHub.sol#L388]

```
386: function collect(uint256 userId, IERC20 erc20)
387:     public
388:     whenNotPaused
389:     onlyDriver(userId)
390:     returns (uint128 amt)
391: {
392:     amt = Splits._collect(userId, _assetId(erc20));
393:     _decreaseTotalBalance(erc20, amt);
394:     erc20.safeTransfer(msg.sender, amt);
395: }
```

Recommended Mitigation Steps

Consider removing the `whenNotPaused` modifier to allow collecting funds while the `DripsHub` contract is paused.

[L-05] An immutable split is unable to collect funds if it has itself set as a split receiver

An immutable split cannot collect funds if it has itself set as a split receiver. This is because the `ImmutableSplitsDriver` contract lacks the functionality to collect available funds.

Findings

[ImmutableSplitsDriver.sol#L66]()

```
53: function createSplits(SplitsReceiver[] calldata receivers, U
54:     public
55:     whenNotPaused
56:     returns (uint256 userId)
57: {
58:     userId = nextUserId();
59:     StorageSlot.getUint256Slot(_counterSlot).value++;
60:     uint256 weightSum = 0;
61:     for (uint256 i = 0; i < receivers.length; i++) {
62:         weightSum += receivers[i].weight;
63:     }
64:     require(weightSum == totalSplitsWeight, "Invalid total r
65:     emit CreatedSplits(userId, dripsHub.hashSplits(receivers
66:     dripsHub.setSplits(userId, receivers);
67:     if (userMetadata.length > 0) dripsHub.emitUserMetadata(u
68: }
```

Recommended Mitigation Steps

Consider preventing setting one of the receivers to the user ID of the newly created immutable split.

[N-01] The NatSpec comment of the `DripsConfigImpl.lt` function neglects to sorting of `dripId`

According to the NatSpec comment of the `DripsConfigImpl.lt` function, the `DripsConfig` struct is sorted by `amtPerSec`, then `start` and then `duration`. However, the `dripId` is not mentioned in the comment:

> /// First compares their `amtPerSec` s, then their `start` s and then their `duration` s.

Findings

[Drips.sol#L93]

```
92: /// @notice Compares two `DripsConfig`s.
93: /// First compares their `amtPerSec`s, then their `start`s a
94: function lt(DripsConfig config, DripsConfig otherConfig) int
95:     return DripsConfig.unwrap(config) < DripsConfig.unwrap(o
96: }
```

[Drips.sol#L1069]

```
1061: function _isOrdered(DripsReceiver memory prev, DripsReceiv
1062:     private
1063:     pure
1064:     returns (bool)
1065: {
1066:     if (prev.userId != next.userId) {
1067:         return prev.userId < next.userId;
1068:     }
1069:     return prev.config.lt(next.config); // @audit-info `dr
1070: }
```

Recommended Mitigation Steps

Consider updating the NatSpec comment to include the `dripId` in the sorting order.

[Alex the Entreprenerd (judge) commented]:

> *(Note: See [original submission] for judge's full commentary.)*

> 9 Low, 1 Refactoring +3

> (includes downgraded findings, see: #274, #278, #280, #281)

> Summing up all findings, this report is by far the most interesting, well done!

Gas Optimizations

For this contest, 11 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **descharre** received the top score from the judge.

The following wardens also submitted reports: NoamYakov, Deivitto, Aymen0909, 0xSmartContract, matrix_0wl, cryptostellar5, 0xA5DF, ReyAdmirado, Rolezn, *and* chaduke .

Gas Optimizations Summary

ID	Finding	Gas saved	Instances
G-01	Make for loop unchecked	644	13
G-02	Use an unchecked block when operands can't underflow/overflow	688	7
G-03	Write element of storage struct to memory when used more than once	10	1
G-04	Call block.timestamp direclty instead of function	22	1
G-05	Make 3 event parameters indexed when possible	1059	2
G-06	Transfer erc20 immediately to Dripshub	57824	1
G-07	Transfer ERC20 immediately to the user	22112	1
G-08	Use double if statements instead of &&	4	40
G-09	Miscellaneous	100	3

[G-01] Make for loop unchecked

The risk of for loops getting overflowed is extremely low. Because it always increments by 1 and is limited to the arrays length. Even if the arrays are extremely long, it will take a massive amount of time and gas to let the for loop overflow.

- Caller.sol#L196-L199: `callBatched()` gas saved: 76
- ImmutableSplitsDriver.sol#L61-L63: `createSplits()` gas saved: 92
- Drips.sol#L247-L249: `DripsHub: receiveDrips()` gas saved: 58
- Drips.sol#L287-L291: `DripsHub: receiveDripsResult()` gas saved: 54
- Drips.sol#L357-L364: `DripsHub: squeezeDrips()` gas saved: 32
- Drips.sol#L450-L459: `DripsHub: squeezeDripsResult()` gas saved: 71
- Drips.sol#L490-L497: `DripsHub: squeezeDripsResult()` gas saved: 29
- Drips.sol#L563-L573: `DripsHub: balanceAt()` gas saved: 59
- Drips.sol#L662-L668: `DripsHub: setDrips()` gas saved: 31
- DripsHub.sol#L613-L616: `emitUserMetadata()` gas saved: 59
- Splits.sol#L127-L129: `DripsHub: splitResult()` gas saved: 10
- Splits.sol#L158-L166: `DripsHub: split()` gas saved: 10
- Splits.sol#L231-L243: `DripsHub: setSplits()` gas saved: 63

There are 2 ways to make a for loop unchecked in a safe way:

```
-       for (uint256 i = 0; i < calls.length; i++) {
-       for (uint256 i = 0; i < calls.length;) {
            Call memory call = calls[i];
            returnData[i] = _call(sender, call.to, call.data, ca
+           unchecked{
+               i++
+           }
        }
```

```
+       function unchecked_inc(uint256 x) private pure returns (
+           unchecked {
+               return x + 1;
+           }
+       }
```

```
-       for (uint256 i = 0; i < calls.length; i++) {
+       for (uint256 i = 0; i < calls.length;  i = unchecked_inc
            Call memory call = calls[i];
            returnData[i] = _call(sender, call.to, call.data, ca
        }
```

[G-02] Use an unchecked block when operands can't underflow/overflow

Drips.sol#L480

Division can't overflow or underflow unless the divisor is -1. Which is not the case here. `DripsHub: squeezeDripsResult()` gas saved: 60

```
        uint256 idxMid = (idx + idxCap) / 2;
```

Drips.sol#L655

There is no possibility of overflowing when incrementing by one. currCycleConfigs is a uint32 but even for that will take a massive amount of time to make it overflow. `DripsHub: setDrips()` gas saved: 215

```
        state.currCycleConfigs++;
```

For ++, the same applies to:

- Drips.sol#L428 `DripsHub: squeezeDrips()` gas saved: 28
- Drips.sol#L959 and Drips.sol#L962: `DripsHub: setDrips()` gas saved: 35
- DripsHub.sol#L136 `DripsHub: registerDriver()` gas saved: 200

DripsHub.sol#L632: because of the require statement above, it can't overflow. `give()` gas saved: 54

DripsHub.sol#L636: amount will never be larger than the total balance: `collect()` gas saved: 96

[G-03] Write element of storage struct to memory when used more than once

When a struct contains a nested mapping, it's not possible to save it in memory. But it's possible to save one element of the struct to memory when it's used more than once. `DripsHub: balanceAt()` gas saved: 10

- Drips.sol#L539-L542

```
        DripsState storage state = _dripsStorage().states[assetI
+       uint32 updateTime = state.updateTime;
-       require(timestamp >= state.updateTime, "Timestamp before
+       require(timestamp >= updateTime, "Timestamp before last 
        require(_hashDrips(receivers) == state.dripsHash, "Inval
-       return _balanceAt(state.balance, state.updateTime, state
+       return _balanceAt(state.balance, updateTime, state.maxEn
```

[G-04] Call block.timestamp direclty instead of function

The `_currTimestamp()` function casts the `block.timestamp` to a uint32. However it's not always necessary to have a uint32.

In the example below you are assigning the timestamp to a uint256. Which makes it unnecessary to cast the timestamp to uint32. So it's better to call `block.timestamp` directly.

Drips.sol#L689: `DripsHub: setDrips()` gas saved: 22

```
-    uint256 enoughEnd = _currTimestamp();
+    uint256 enoughEnd = block.timestamp;
```

[G-05] Make 3 event parameters indexed when possible

It's the most gas efficient to make up to 3 event parameters indexed. If there are less than 3 parameters, you need to make all parameters indexed.

- DripsHub.sol#L93

- event UserMetadataEmitted(uint256 indexed userId, bytes32 indexed key, bytes value);
- event UserMetadataEmitted(uint256 indexed userId, bytes32 indexed key, bytes indexed value);

```
    The event is used in the function emitUserMetaData, this function is us
```

- DripsHub.sol: `emitUserMetadata()` gas saved: 364
- AddressDriver.sol: `emitUserMetadata()` gas saved: 208
- ImmutableSplitsDriver.sol: `createSplits()` gas saved: 139
- NFTDriver.sol: `emitUserMetadata()` gas saved: 139
- NFTDriver.sol: `safeMint()` gas saved: 209

The same extra indexed parameter can be applied to:

- Drips.sol#L153 `DripsHub: setDrips()`

[G-06] Transfer erc20 immediately to Dripshub

- `Dripshub: give()` gas saved: 9449
- `AddressDriver: give()` gas saved: 29439
- `NFTDriver: give()` gas saved: 18936

When you call the `give()` function in the Address or NFTDriver. The erc20 token are first getting send to those contracts and afterwards to the DripsHub contract. It's also possible to send the tokens directly to the dripsHub contract.

AddressDriver.sol#L170-L176 NFTDriver.sol#L285-L291

```
    function _transferFromCaller(IERC20 erc20, uint128 amt) inte
-        erc20.safeTransferFrom(_msgSender(), address(this), amt)
+        erc20.safeTransferFrom(_msgSender(), address(dripsHub), 
         // Approval is done only on the first usage of the ERC-2
```

```
-           if (erc20.allowance(address(this), address(dripsHub)) ==
-               erc20.safeApprove(address(dripsHub), type(uint256).m
            }
        }
```

DripsHub.sol#L417

```
        function give(uint256 userId, uint256 receiver, IERC20 erc20
            public
            whenNotPaused
            onlyDriver(userId)
        {
            _increaseTotalBalance(erc20, amt);
            Splits._give(userId, receiver, _assetId(erc20), amt);
-           erc20.safeTransferFrom(msg.sender, address(this), amt);
        }
```

setDrips need to be adjusted or you can create a seperate function for that.

[G-07] Transfer ERC20 immediately to the user

- `AddressDriver: collect()` gas saved: 12954
- `NFTDriver: collect()` gas saved: 9158

The same thing can be done for the `collect()` function. Instead of transferring first to the address or NFTdriver. You can instantly transfer to the user.
DripsHub.sol#L386-L395)

```
-       function collect(uint256 userId, IERC20 erc20)
+       function collect(uint256 userId, IERC20 erc20, address trans
            public
            whenNotPaused
            onlyDriver(userId)
            returns (uint128 amt)
        {
            amt = Splits._collect(userId, _assetId(erc20));
            _decreaseTotalBalance(erc20, amt);
-           erc20.safeTransfer(msg.sender, amt);
+           erc20.safeTransfer(transferTo, amt);
```

```
    }
```

AddressDriver.sol#L60-L63

```diff
      function collect(IERC20 erc20, address transferTo) public wh
-         amt = dripsHub.collect(callerUserId(), erc20);
+         amt = dripsHub.collect(callerUserId(), erc20, transferTo
-         erc20.safeTransfer(transferTo, amt);
      }
```

[G-08] Use double if statements instead of &&

If the if statement has a logical AND and is not followed by an else statement, it can be replaced with 2 if statements.

- Drips.sol#L700
- Drips.sol#L709
- Drips.sol#L909
- Drips.sol#L929

[G-09] Miscellaneous

Don't call a function when initializing an immutable variable

Saves a little bit of deployment gas

```diff
-   bytes32 private immutable _counterSlot = _erc1967Slot("eip19
+   bytes32 private immutable _counterSlot = bytes32(uint256(kec
```

Use a mapping type of a struct directly instead of assigning it to another storage variable

- Drips.sol#L246-L254

- mapping(uint32 => AmtDelta) storage amtDeltas = state.amtDeltas; for (uint32 cycle = fromCycle; cycle < toCycle; cycle++) {
- delete amtDeltas[cycle];
- delete state.amtDeltas[cycle]; } // The next cycle delta must be relative to the last received cycle, which got zeroed. // In other words the next cycle delta must be an absolute value. if (finalAmtPerCycle != 0) {
- amtDeltas[toCycle].thisCycle += finalAmtPerCycle;
- state.amtDeltas[toCycle].thisCycle += finalAmtPerCycle; }

```
### If statement can be adjusted
In the `_receiveDrips()` function you can change the check to `received
```

Drips.sol#L243 `receiveDrips()` gas saved: 75

```diff
        (receivedAmt, receivableCycles, fromCycle, toCycle, fina
            _receiveDripsResult(userId, assetId, maxCycles);
-       if (fromCycle != toCycle) {
+       if (receivedAmt != 0) {
```

Alex the Entreprenerd (judge) commented:

> Best because the savings are high impact and tangible.

> [G-01] Make for loop unchecked 644 13

> Let's say 260 gas (13 * 20)

> [G-02] Use an unchecked block when operands can't underflow/overflow 688 7

> 20 * 7 = 140

> [G-03] Write element of storage struct to memory when used more than once 10 1

> 100 gas

> [G-04] Call block.timestamp direclty instead of function 22 1

> 16

> [G-06] Transfer erc20 immediately to Dripshub 57824 1

> 5k

> [G-07] Transfer ERC20 immediately to the user 22112 1

> 5k

> Rest is marginal

> 10k+

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top