Optimism Drippie Security Review

Auditors

Noah Marconi, Lead Security Researcher

Emanuele Ricci, Security Researcher

Hrishikesh Bhat, Apprentice

0xNazgul, Apprentice

Report prepared by: Pablo Misirov & 0xNazgul

October 4, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Drippie is a system for managing automated contract interactions. A specific interaction is called a "drip" and can be executed according to some condition (called a dripcheck) and an execution interval. Drips cannot be executed faster than the execution interval. Drips can trigger arbitrary contract calls where the calling contract is this contract address. Drips can also send ETH value, which makes them ideal for keeping addresses sufficiently funded with ETH. Drippie is designed to be connected with smart contract automation services so that drips can be executed automatically. However, Drippie is specifically designed to be separated from these services so that trust assumptions are better compartmentalized.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Optimism Drippie according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Optimism engaged with Spearbit to review Drippie. In this period of time a total of 34 issues were found.

Summary

Project Name	Drippie
Repository	Drippie
Commit	2a7be367634f14773...
Type of Project	L2, Automation
Audit Timeline	Aug 8st - Aug 17th
Methods	Manual Review, Mythril

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	11
Gas Optimizations	5
Informational	17
Total Issues	34

5 Findings

5.1 Medium Risk

5.1.1 Permitting Multiple Drip Calls Per Block

Severity: *Medium Risk*

Context: Drippie.sol#L266

Description: The inline comments correctly note that reentrancy is possible and permitted when `state.config.interval` is 0. We are currently unaware of use cases where this is desirable.

Reentrancy is one risk, flashbot bundles are a similar risk where the drip may be called multiple times by the same actor in a single block. A malicious actor may abuse this ability, especially if interval is misconfigured as 0 due to JavaScript type coercion.

A reentrant call or flashbot bundle may be used to frontrun an owner attempting to archive a drip or attempting to withdraw assets.

Recommendation: We recommend limiting drip calls to 1 per block.

Document the transaction order dependence (frontrunning) risk for owners wishing to archive a drip. Reasonable drip intervals can be employed to prevent this attack.

If it is important to permit multiple calls to the same drip in a single block, we recommend making the behavior opt-in rather than default if no `state.config.interval` specified.

```
+function create(string memory _name, DripConfig memory _config, bool allowMultiplePerBlock) external
↪  onlyOwner {
-function create(string memory _name, DripConfig memory _config) external onlyOwner {
    // Make sure this drip doesn't already exist. We *must* guarantee that no other function
    // will ever set the status of a drip back to NONE after it's been created. This is why
    // archival is a separate status.
    require(
        drips[_name].status == DripStatus.NONE,
        "Drippie: drip with that name already exists"
    );

+    require(
+        _config.interval > 0 || allowMultiplePerBlock,
+        "Drippie, explict opt-in for 0 interval"
+    );

    // We initialize this way because Solidity won't let us copy arrays into storage yet.
    DripState storage state = drips[_name];
    state.status = DripStatus.PAUSED;
    state.config.interval = _config.interval;
    state.config.dripcheck = _config.dripcheck;
    state.config.checkparams = _config.checkparams;

    // Solidity doesn't let us copy arrays into storage, so we push each array one by one.
    for (uint256 i = 0; i < _config.actions.length; i++) {
        state.config.actions.push(_config.actions[i]);
    }

    // Tell the world!
    emit DripCreated(_name, _name, _config);
}
```

Optimism: Fixed in PR #3280.

Spearbit: Fixed.

5.2 Low Risk

5.2.1 Version Bump to Latest

Severity: *Low Risk*

Context: Drippie.sol#L2, CheckBalanceHigh.sol#L2, CheckBalanceLow.sol#L2, CheckGelatoLow.sol#L2, CheckTrue.sol#L2

Description: During the review, a new version of solidity was released with an important bugfix.

Recommendation: Move from `0.8.15` to `0.8.16`

Optimism: Fixed here PR #3567.

Spearbit:

- `Drippie` solidity version has been updated to `0.8.16` in PR#3280
- `CheckBalanceHigh.sol`, `CheckBalanceLow.sol`, `CheckGelatoLow.sol`, `CheckTrue.sol` solidity version have been updated to `0.8.16` in PR#3567

5.2.2 DOS from External Calls in `Drippie.executable` / `Drippie.drip`

Severity: *Low Risk*

Context: Drippie.sol#L233-L236, Drippie.sol#L284

Description: In both the `executable` and `drip` (which also calls `executable`) functions, the `Drippie` contract interacts with some external contract via low-level calls.

The external call could revert or fail with an Out of Gas exception causing the entire drip to fail.

The severity is low beacuse in the case where a `drip` reverts due to a misconfigured or malicious `dripcheck` or `target`, the `drip` can still be `archived` and a new one can be created by the owner.

Recommendation: The DOS vector is documented inline, consider elevating to @natspec and user docs.

Worth noting is that drips expect the `DripAction.target` to revert on failure and not fail silently. In other words, all action targets MUST revert on failure, else unexpected behaviour between actions may occur.

5.2.3 Use `call.value` over `transfer` in `withdrawETH`

Severity: *Low Risk*

Context: AssetReceiver.sol#L89

Description: `transfer` is no longer recommended as a default due to unpredictable gas cost changes in future evm hard forks (see here for more background.)

While useful to use `transfer` in some cases (such as sending to EOA or contract which does not process data in the fallback or receiver functions), this particular contract does not benefit: `withdrawETH` is already `owner` gated and is not at risk of reentrancy as `owner` already has permission to drain the contract's ether in a single call should they choose.

Recommendation: Use `call.value` over `transfer` in `withdrawETH`.

Spearbit: Note that the implementation in PR#3280 has moved from `transfer` to a low-level `call` but has still some issues.

The `success` value returned from the `call` is never used, returned as function `returned` variable or checked at all. Because of this reason, the current implementation of `withdrawETH` allow the function to "fail silently" when the internal low-level `call` revert and return `success = false`. In this case, the ETH has not been transferred to the recipient, but because the "main" transaction does not revert the `WithdrewETH` is still emitted, and the caller will think that the ETH has been correctly transferred.

Optimism: We agree and intend to fix this, though not immediately. This is OK as it's an `onlyOwner` gated function which we expect to use very infrequently.

Spearbit: Acknowledged.

5.2.4 Input Validation Checks for `Drippie.create`

Severity: *Low Risk*

Context: Drippie.sol#L126-L149

Description: `Drippie.create` does not validate input potentially leading to unintended results.

The function should check:

- `_name` is not an empty string to avoid creating drip that would be able to read on frontend UI.
- `_config.dripcheck` should not be `address(0)` otherwise `executable` will always revert.
- `_config.actions.length` should be at least one (`_config.actions.length > 0`) to prevent creating drips that do nothing when executed.
- `DripAction.target` should not be `address(0)` to prevent burning ETH or interacting with the zero address during drip's execution.

Recommendation: Consider implementing the suggested checks to prevent misconfigured drips.

`DripAction.data` and `DripConfig.params` are not type checked, however, there is no simple fix without sacrificing flexibility. We recommend surfacing a caution in user docs.

5.2.5 Ownership Initialization and Transfer Safety on `Owned.setOwner`

Severity: *Low Risk*

Context: Drippie.sol#L116

Description: Consider the following scenarios.

- Scenario 1

`Drippie` allows the `owner` to be both initialized and set to `address(0)`. If this scenario happens nobody will be able to manage the `Drippie` contract, thus preventing any of the following operations:

- Creating a new drip
- Updating a drip's status (pausing, activating or archiving a drip)

If set to the zero address, all the `onlyOwner` operations in `AssetReceiver` and `Transactor` will be uncallable.

This scenario where the owner can be set to `address(0)` can occur when `address(0)` is passed to the `constructor` or `setOwner`.

- Scenario 2

`owner` may be set to `address(this)`. Given the static nature of `DripAction.target` and `DripAction.data` there is no benefit of setting owner to `address(this)`, and all instances can be assumed to have been done so in error.

Recommendation: Add a check for `address(0)` and `address(this)` in both the `constructor` and `setOwner`.

For added safety, require the new owner to accept the ownership before ownership is transferred.

5.2.6 Unchecked Return and Handling of Non-standard Tokens in `AssetReceiver`

Severity: *Low Risk*

Context: AssetReceiver.sol#L116

Description: The current `AssetReceiver` contract implement "direct" ETH and ERC20 token transfers, but does not cover edge cases like non-standard ERC20 tokens that do not:

- revert on failed transfers
- adhere to ERC20 interface (i.e. no return value)

An ERC20 token that does not revert on failure would cause the `WithdrewERC20` event to emit even though no transfer took place.

An ERC20 token that does not have a return value will revert even if the call would have otherwise been successful.

Solmate libraries already used inside the project offer a utility library called SafeTransferLib.sol which covers such edge cases.

Be aware of the developer comments in the natspec:

> /// @dev Use with caution! Some functions in this library knowingly create dirty bits at the destination of the free memory pointer. /// @dev Note that none of the functions in this library check that a token has code at all! That responsibility is delegated to the caller.

Recommendation: Consider integrating Solmate `SafeTransferLib` inside `AssetReceiver` to cover edge cases.

5.2.7 `AssetReceiver` Allows Burning ETH, ERC20 and ERC721 Tokens

Severity: *Low Risk*

Context: AssetReceiver.sol#L89, AssetReceiver.sol#L116, AssetReceiver.sol#L133

Description: `AssetReceiver` contains functions that allow the `owner` of the contract to withdraw ETH, ERC20 and ERC721 tokens.

Those functions allow specifying the `receiver` address of ETH, ERC20 and ERC721 tokens but they do not check that the receiver address is not `address(0)`.

By not doing so, those functions allow to:

- Burn ETH if sent to `address(0)`.
- Burn ERC20 tokens if sent to `address(0)` and the ERC20 `_asset` allow tokens to be burned via `transfer` (For example, Solmate's ERC20 allow that, OpenZeppelin instead will revert if the recipient is `address(0)`).
- Burn ERC721 tokens if sent to `address(0)` and the ERC721 `_asset` allow tokens to be burned via `transferFrom` (For example, both Solmate and OpenZeppelin implementations prevent to send the `_id` to the `address(0)` but you don't know if that is still true about custom ERC721 contract that does not use those libraries).

Recommendation: Add a check on all those functions to revert if `_to` is `address(0)`.

5.2.8 `AssetReceiver` Not Implementing `onERC721Received` Callback Required by `safeTransferFrom`.

Severity: *Low Risk*

Context: AssetReceiver.sol

Description: `AssetReceiver` contains the function `withdrawERC721` that allow the `owner` to withdraw ERC721 tokens.

As stated in the EIP-721, the `safeTransferFrom` (used by the sender to transfer ERC721 tokens to the `AssetReceiver`) will revert if the `target` contract (`AssetReceiver` in this case) is not implementing `onERC721Received` and returning the expected value `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`.

Recommendation: Add the `onERC721Received` callback function in the `AssetReceiver` contract to be able to receive ERC721 tokens.

5.2.9 Both `Transactor.CALL` and `Transactor.DELEGATECALL` Do Not Emit Events

Severity: *Low Risk*

Context: Transactor.sol#L27-L34, Transactor.sol#L46-L53

Description: `Transactor` contains a "general purpose" `DELEGATECALL` and `CALL` function that allow the owner to execute a `delegatecall` and `call` toward a `target` address passing an arbitrary payload.

Both of those functions are executing `delegatecall` and `call` without emitting any events. Because of the general-purpose nature of these function, it would be considered a good security measure to `emit` events to track the function's usage. Those events could be then used to monitor and track usage by external monitoring services.

Recommendation: Consider adding event emission to both `delegatecal` and `call` functions.

5.2.10 Both `Transactor.CALL` and `Transactor.DELEGATECALL` Do Not Check the Result of the Execution

Severity: *Low Risk*

Context: Transactor.sol#L27-L34, Transactor.sol#L46-L53

Description: The `Transactor` contract contains a "general purpose" `DELEGATECALL` and `CALL` function that allow the owner to execute a `delegatecall` and `call` toward a `target` address passing an arbitrary payload.

Both functions return the `delegatecall` and `call` result back to the caller without checking whether execution was successful or not.

By not implementing such check, the transaction could fail silently. Another side effect is that the ETH sent along with the execution (both functions are `payable`) would remain in the `Drippie` contract and not transferred to the `_target`.

Test example showcasing the issue:

```
contract Useless {
    // A contract that have no functions
    // No fallback functions
    // Will not accept ETH (only from selfdestruct/coinbase)
}
```

```
function test_transactorCALL() public {
    Useless useless = new Useless();
    bool success;

    vm.deal(deployer, 3 ether);
    vm.deal(address(drippie), 0 ether);
    vm.deal(address(useless), 0 ether);

    vm.prank(deployer);
    // send 1 ether via `call` to a contract that cannot receive them
```

```
    (success, ) = drippie.CALL{value: 1 ether}(address(useless), "", 100000, 1 ether);
    assertEq(success, false);

    vm.prank(deployer);
    // Perform a `call` to a not existing target's function
    (success, ) = drippie.CALL{value: 1 ether}(address(useless),
↪   abi.encodeWithSignature("notExistingFn()"), 100000, 1 ether);
    assertEq(success, false);

    assertEq(deployer.balance, 1 ether);
    assertEq(address(drippie).balance, 2 ether);
    assertEq(address(useless).balance, 0);
}

function test_transactorDELEGATECALL() public {
    Useless useless = new Useless();
    bool success;


    vm.deal(deployer, 3 ether);
    vm.deal(address(drippie), 0 ether);
    vm.deal(address(useless), 0 ether);

    vm.prank(deployer);
    // send 1 ether via `delegatecall` to a contract that cannot receive them
    (success, ) = drippie.DELEGATECALL{value: 1 ether}(address(useless), "", 100000);
    assertEq(success, false);

    vm.prank(deployer);
    // Perform a `delegatecall` to a not existing target's function
    (success, ) = drippie.DELEGATECALL{value: 1 ether}(address(useless),
↪   abi.encodeWithSignature("notExistingFn()"), 100000);
    assertEq(success, false);

    assertEq(deployer.balance, 1 ether);
    assertEq(address(drippie).balance, 2 ether);
    assertEq(address(useless).balance, 0);
}
```

Recommendation: Consider adding a check on both functions to cause the transaction to revert in case the execution of `delegatecall` or `call` returns `success == false`.

5.2.11 `Transactor.DELEGATECALL` Data Overwrite and `selfdestruct` Risks

Severity: *Low Risk*

Context: Transactor.sol#L46-L53

Description: The `Transactor` contract contains a "general purpose" `DELEGATECALL` function that allow the owner to execute a `delegatecall` toward a `target` address passing an arbitrary payload.

Consider the following scenarios:

- Scenario 1

A malicious target contract could `selfdestruct` the `Transactor` contract and as a consequence the contract that is inheriting from `Transactor`.

Test example showcasing the issue:

```
contract SelfDestroyer {
    function destroy(address receiver) external {
        selfdestruct(payable(receiver));
    }
}
```

```
function test_canOwnerSelftDestructDrippie() public {
    // Assert that Drippie exist
    assertStatus(DEFAULT_DRIP_NAME, Drippie.DripStatus.PAUSED);
    assertGt(getContractSize(address(drippie)), 0);

    // set it to active
    vm.prank(deployer);
    drippie.status(DEFAULT_DRIP_NAME, Drippie.DripStatus.ACTIVE);
    assertStatus(DEFAULT_DRIP_NAME, Drippie.DripStatus.ACTIVE);

    // fund the drippie with 1 ETH
    vm.deal(address(drippie), 1 ether);

    uint256 deployerBalanceBefore = deployer.balance;
    uint256 drippieBalanceBefore = address(drippie).balance;

    // deploy the destroyer
    SelfDestroyer selfDestroyer = new SelfDestroyer();

    vm.prank(deployer);
    drippie.DELEGATECALL(address(selfDestroyer), abi.encodeWithSignature("destroy(address)", deployer),
↪   gasleft());


    uint256 deployerBalanceAfter = deployer.balance;
    uint256 drippieBalanceAfter = address(drippie).balance;

    // assert that the deployer has received the balance that was present in Drippie
    assertEq(deployerBalanceAfter, deployerBalanceBefore + drippieBalanceBefore);
    assertEq(drippieBalanceAfter, 0);

    // Weird things happens with forge
    // Because we are in the same block the code of the contract is still > 0 so
    // Cannot use assertEq(getContractSize(address(drippie)), 0);
    // Known forge issue
    // 1) Forge resets storage var to 0 after self-destruct (before tx ends) 2654 ->
↪   https://github.com/foundry-rs/foundry/issues/2654
    // 2) selfdestruct has no effect in test 1543 -> https://github.com/foundry-rs/foundry/issues/1543

    assertStatus(DEFAULT_DRIP_NAME, Drippie.DripStatus.PAUSED);

}
```

- Scenario 2

The `delegatecall` allows the owner to intentionally, or accidentally, overwrite the content of the `drips` mapping. By being able to modify the `drips` mapping, a malicious user would be able to execute a series of actions like:

Changing drip's status:

- Activating an archived `drip`
- Deleting a drip by changing the status to NONE (this allows the owner to override entirely the drip by calling again `create`)
- Switching an active/paused drip to paused/active

- etc..

Change drip's interval:

- Prevent a `drip` from being executed any more by setting `interval` to a very high value
- Allow a `drip` to be executed more frequently by lowering the `interval` value
- Enable reentrancy by setting `interval` to `0`

Change drip's actions:

- Override an action to send drip's contract balance to an arbitrary address
- etc..

Test example showcasing the issue:

```
contract ChangeDrip {
    address public owner;
    mapping(string => Drippie.DripState) public drips;

    function someInnocentFunction() external {
        drips["FUND_BRIDGE_WALLET"].config.actions[0] = Drippie.DripAction({
            target: payable(address(1024)),
            data: new bytes(0),
            value: 1 ether
        });
    }
}
```

```
function test_canDELEGATECALLAllowReplaceAction() public {
    vm.deal(address(drippie), 10 ether);
    vm.deal(address(attacker), 0 ether);

    // Create an action with name "FUND_BRIDGE_WALLET" that have the function
    // To fund a wallet
    vm.startPrank(deployer);
    string memory fundBridgeWalletName = "FUND_BRIDGE_WALLET";
    Drippie.DripAction[] memory actions = new Drippie.DripAction[](1);

    // The first action will send Bob 1 ether
    actions[0] = Drippie.DripAction({
        target: payable(address(alice)),
        data: new bytes(0),
        value: 1 ether
    });
    Drippie.DripConfig memory config = createConfig(100, IDripCheck(address(checkTrue)), new bytes(0),
↪   actions);
    drippie.create(fundBridgeWalletName, config);
    drippie.status(fundBridgeWalletName, Drippie.DripStatus.ACTIVE);
    vm.stopPrank();

    // Deploy the malicius contract
    vm.prank(attacker);
    ChangeDrip changeDripContract = new ChangeDrip();

    // make the owner of drippie call via DELEGATECALL an innocentfunction of the exploiter contract
    vm.prank(deployer);
    drippie.DELEGATECALL(address(changeDripContract),
↪   abi.encodeWithSignature("someInnocentFunction()"), 1000000);

    // Now the drip action should have changed, anyone can execute it and funds would be sent to
    // the attacker and not to the bridge wallet
    drippie.drip(fundBridgeWalletName);

    // Assert we have drained Drippie
    assertEq(attacker.balance, 1 ether);
    assertEq(address(drippie).balance, 9 ether);
}
```

- Scenario 3

Calling a malicious contract or accidentally calling a contract which does not account for `Drippie`'s storage layout can result in `owner` being overwritten.

Test example showcasing the issue:

```
contract GainOwnership {
    address public owner;

    function someInnocentFunction() external {
        owner = address(1024);
    }
}
```

```
function test_canDELEGATECALLAllowOwnerLoseOwnership() public {
    vm.deal(address(drippie), 10 ether);
    vm.deal(address(attacker), 0 ether);

    // Deploy the malicius contract
    vm.prank(attacker);
    GainOwnership gainOwnershipContract = new GainOwnership();

    // make the owner of drippie call via DELEGATECALL an innocentfunction of the exploiter contract
    vm.prank(deployer);
    drippie.DELEGATECALL(address(gainOwnershipContract),
↪   abi.encodeWithSignature("someInnocentFunction()"), 1000000);

    // Assert that the attacker has gained onwership
    assertEq(drippie.owner(), attacker);

    // Steal all the funds
    vm.prank(attacker);
    drippie.withdrawETH(payable(attacker));

    // Assert we have drained Drippie
    assertEq(attacker.balance, 10 ether);
    assertEq(address(drippie).balance, 0 ether);
}
```

Recommendation:

- On all Scenarios

Document and instruct users to pay attention to each contract that `Transactor` interacts with when `DELEGATECALL` is called to prevent situations like this.

If `Drippie` does not need to allow the `owner` to execute general purpose `delegatecall` consider removing the `Transactor` dependency from the inheritance chain.

Be particularly cautious calling out to upgradeable contracts/proxies.

- Scenario 1

Deploying with `create2` allows for recovery of any tokens managed by the `Drippie` contract in the event it is accidentally `selfdestruct`ed by redeploying to the same address.

- Scenario 3

A postcondition can also assist in protecting accidental owner overwriting:

```
function DELEGATECALL(
    address _target,
    bytes memory _data,
    uint256 _gas
) external payable onlyOwner returns (bool, bytes memory) {
+    address prevOwner = owner;
    // slither-disable-next-line controlled-delegatecall
-    return _target.delegatecall{ gas: _gas }(_data);
+    (bool success, bytes memory res) = _target.delegatecall{ gas: _gas }(_data);
+    require(prevOwner == owner, "accidential onwer overwrite");
+    return (success, res);
}
```

5.3 Gas Optimization

5.3.1 Use `calldata` over `memory`

Severity: *Gas Optimization*

Context: Drippie.sol#L126, Drippie.sol#L160, Drippie.sol#L213, Drippie.sol#L252

Description: Some gas savings if function arguments are passed as `calldata` instead of `memory`.

Recommendation: Use `calldata` in these instances.

Optimism: Addressed in PR#3280.

Spearbit: Fixed.

5.3.2 Avoid String `name`s in Events and Mapping Key

Severity: *Gas Optimization*

Context: Drippie.sol#L111

Description: Drip events emit an indexed `nameref` and the `name` as a string. These strings must be passed into every `drip` call adding to gas costs for larger strings.

Recommendation: For off chain uses, i.e. user interface display, the names are useful. A more gas efficient approach would be to use `uint256` or `bytes32` for drip mapping keys.

The string `name`s may still be stored in `DripState` for off chain reading, but gas would be saved in not logging names each time a drip is called and not needing to incur the variable gas costs longer names introduce.

5.3.3 Avoid Extra `sload`s on `Drippie.status`

Severity: *Gas Optimization*

Context: Drippie.sol#L160

Description: Information for emitting event can be taken from calldata instead of reading from storage.

Can skip repeat `drips[_name].status` reads from storage.

Recommendation: Consider implementing the following fixes.

```
function status(string memory _name, DripStatus _status) external onlyOwner {

    ...snip...

    // If we made it here then we can safely update the status.
    drips[_name].status = _status;
+    emit DripStatusUpdated(_name, _name, _status);
-    emit DripStatusUpdated(_name, _name, drips[_name].status);

}
```

and

```
function status(string memory _name, DripStatus _status) external onlyOwner {

    ...snip...

+    DripStatus currentStatus = drips[_name].status;

    // Make sure the drip in question actually exists. Not strictly necessary but there doesn't
    // seem to be any clear reason why you would want to do this, and it may save some gas in
    // the case of a front-end bug.
    require(
```

```
+        currentStatus != DripStatus.NONE,
-        drips[_name].status != DripStatus.NONE,
         "Drippie: drip with that name does not exist"
     );

     // Once a drip has been archived, it cannot be un-archived. This is, after all, the entire
     // point of archiving a drip.
     require(
+        currentStatus != DripStatus.ARCHIVED,
-        drips[_name].status != DripStatus.ARCHIVED,
         "Drippie: drip with that name has been archived"
     );

     // Although not strictly necessary, we make sure that the status here is actually changing.
     // This may save the client some gas if there's a front-end bug and the user accidentally
     // tries to "change" the status to the same value as before.
     require(
+        currentStatus != _status,
-        drips[_name].status != _status,
         "Drippie: cannot set drip status to same status as before"
     );

     // If the user is trying to archive this drip, make sure the drip has been paused. We do
     // not allow users to archive active drips so that the effects of this action are more
     // abundantly clear.
     if (_status == DripStatus.ARCHIVED) {
         require(
+            currentStatus == DripStatus.PAUSED,
-            drips[_name].status == DripStatus.PAUSED,
             "Drippie: drip must be paused to be archived"
         );
     }

     ...snip...
}
```

Optimism: Addressed in PR#3280.

Spearbit: Fixed.

5.3.4 Use Custom Errors Instead of Strings

Severity: *Gas Optimization*

Context: Drippie.sol

Description: To save some gas the use of custom errors leads to cheaper deploy time cost and run time cost. The run time cost is only relevant when the revert condition is met.

Recommendation: Consider using custom errors instead of revert strings.

Optimism: Chose not to implement.

Spearbit: Acknowledged.

5.3.5 Increment In The For Loop Post Condition In An Unchecked Block

Severity: *Gas Optimization*

Context: Drippie.sol#L143, Drippie.sol#L273

Description: This is only relevant if you are using the default solidity checked arithmetic. `i++` involves checked arithmetic, which is not required. This is because the value of `i` is always strictly less than length <= 2**256 - 1. Therefore, the theoretical maximum value of `i` to enter the for-loop body is `2**256 - 2`. This means that the `i++` in the for loop can never overflow. Regardless, the overflow checks are performed by the compiler.

Unfortunately, the Solidity optimizer is not smart enough to detect this and remove the checks. One can manually do this by:

```
for (uint i = 0; i < length; ) {
    // do something that doesn't change the value of i
    unchecked {
        ++i;
    }
}
```

Recommendation: Consider doing the increment in the for loop post condition in an unchecked block.

Optimism: Chose not to implement.

Spearbit: Acknowledged.

5.4 Informational

5.4.1 `DripState.count` Location and Use

Severity: *Informational*

Context: Drippie.sol#L61, Drippie.sol#L300

Description: `DripState.count` is recorded and never used within the `Drippie` or `IDripCheck` contracts.

`DripState.count` is also incremented after all external calls, inconsistent with Checks, Effects, Interactions convention.

Recommendation: Consider implementing the following recommendations.

Recommendation 1: increment `DripState.count` before external calls.

Recommendation 2: consider removing `DripState.count` entirely if it is not used on chain.

Recommendation 3: if `DripState.count` is not removed, consider whether the information is useful in any future `DripCheck`s.

Optimism: Recommendation 1 implemented in PR#3280.

Spearbit: Recommendation 1 implemented.

5.4.2 Type Checking Foregone on `DripCheck`

Severity: *Informational*

Context: IDripCheck.sol#L4

Description: Passing params as `bytes` makes for a flexible `DripCheck`, however, type checking is lost.

Recommendation: A helper may be added to `DripCheck`s for users to confirm properly constructed params `bytes` array prior to creating a `drip`.

```
contract CheckBalanceLow is IDripCheck {
    event _EventToExposeStructInABI__Params(Params params);
    struct Params {
        address target;
        uint256 threshold;
    }

    function check(bytes memory _params) external view returns (bool) {
        Params memory params = abi.decode(_params, (Params));

        // Check target ETH balance is below threshold.
        return params.target.balance < params.threshold;
    }

+    function encodeCheck(address target, uint256 threshold)
+        external
+        view
+        returns (bytes memory)
+    {
+        Params memory toEncode = Params(target, threshold);
+        return abi.encode(toEncode);
+    }
}
```

5.4.3 Confirm Blind ERC721 Transfers are Intended

Severity: *Informational*

Context: AssetReceiver.sol#L133

Description: `AssetReceiver` uses `transferFrom` instead of `safeTransferFrom`.

The callback on `safeTransferFrom` often poses a reentrancy risk but in this case the function is restricted to `onlyOwner`.

Recommendation: Consider added safety of `safeTransferFrom` when sending to contracts.

5.4.4 Code Contains Empty Blocks

Severity: *Informational*

Context: Transactor.sol#L14, AssetReceiver.sol#L63

Description: It's best practice that when there is an empty block, to add a comment in the block explaining why it's empty. While not technically errors, they can cause confusion when reading code.

Recommendation: Consider adding `/* Comment on why */` to the empty blocks.

Optimism: Chose not to implement.

Spearbit: Acknowledged.

5.4.5 Code Structure Deviates From Best-Practice

Severity: *Informational*

Context: CheckGelatoLow.sol#L15-L20, CheckBalanceLow.sol#L11-L15, CheckBalanceHigh.sol#L11-L15, Drippie.sol#L100-L111

Description: The best-practice layout for a contract should follow this order:

- State variables.
- Events.
- Modifiers.
- Constructor.
- Functions.

Function ordering helps readers identify which functions they can call and find constructor and fallback functions easier. Functions should be grouped according to their visibility and ordered as: constructor, receive function (if exists), fallback function (if exists), external, public, internal, private. Some constructs deviate from this recommended best-practice: structs and mappings after events.

Recommendation: Consider adopting recommended best-practice for code structure and layout.

Optimism: Fixed in PR#3280.

Spearbit: Fixed.

5.4.6 Missing or Incomplete NatSpec

Severity: *Informational*

Context: AssetReceiver.sol#L19, CheckTrue.sol, CheckGelatoLow.sol, CheckBalanceLow.sol, CheckBalanceHigh.sol

Description: Some functions are missing @notice/@dev NatSpec comments for the function, @param for all/some of their parameters and @return for return values. Given that NatSpec is an important part of code documentation, this affects code comprehension, auditability and usability.

Recommendation: Consider adding in full NatSpec comments for all functions to have complete code documentation for future use.

Optimism: Fixed in PR#3280.

Spearbit: Fixed.

5.4.7 Checking Boolean Against Boolean

Severity: *Informational*

Context: Drippie.sol#L256-L259

Description: `executable` returns a boolean in which case the comparison to `true` is unnecessary.

`executable` also reverts if any precondition check fails in which case `false` will never be returned.

Recommendation: If important to check, change `require` to `assert` to facilitate invariant checking (using tools like Mythril):

```
function drip(string memory _name) external {
    DripState storage state = drips[_name];

    // Make sure the drip can be executed.
-    require(
+    assert(
-        executable(_name) == true,
+        executable(_name)
        "Drippie: drip cannot be executed at this time, try again later"
    );

  ...snip...

}
```

Or, remove the extra check entirely:

```
function drip(string memory _name) external {
    DripState storage state = drips[_name];

    // Make sure the drip can be executed.
-    require(
-        executable(_name) == true,
-        "Drippie: drip cannot be executed at this time, try again later"
-    );
+    executable(_name)
  ...snip...

}
```

Optimism: The recommendation has been implemented in the PR#3280

Spearbit: Fixed.

5.4.8 `Drippie.executable` Never Returns `false` Only `true` or Reverts

Severity: *Informational*

Context: Drippie.sol#L206-L240

Description: The `executable` implemented in the `Drippie` contract has the following signature `executable(string memory _name) public view returns (bool)`.

From the signature and the natspec documentation `@return True if the drip is executable, false otherwise.` Without reading the code, a user/developer would expect that the function returns `true` if all the checks passes otherwise `false` but in reality the function will always return `true` or revert.

Because of this behavior, a reverting `drip` that do not pass the requirements inside `executable` will never revert with the message present in the following code executed by the `drip` function

```
require(
    executable(_name) == true,
    "Drippie: drip cannot be executed at this time, try again later"
);
```

Recommendation: If the current implementation is the expected behavior, consider updating the natspec of the function to reflect the function's implementation.

Optimism: The recommendation has been implemented in the PR#3280.

Spearbit: Fixed.

5.4.9 Drippie Use Case Notes

Severity: *Informational*

Description: Drippie intends to support use cases outside of the initial hot EOA top-up use case demonstrated by Optimism. To further clarify, we've noted that drips support:

- Sending eth
- External function calls with fixed params
- Preconditions

Examples include, conditionally transferring eth or tokens. Calling an admin function iff preconditions are met.

Drips do not support:

- Updating the drip contract storage
- Altering params
- Postconditions

Examples include, vesting contracts or executing Uniswap swaps based on recent moving averages (which are not without their own risks). Where dynamic params or internal accounting is needed, a separate contract needs to be paired with the drip.

Recommendation: No changes needed. Documenting only.

5.4.10 Augment Documentation for `dripcheck.check` Indicating Precondition Check Only Performed

Severity: *Informational*

Context: Drippie.sol#L252-L302

Description: Before executing the whole batch of `actions` the `drip` function call `executable` that check if the drip can be executed. Inside `executable` an external contract is called by this instruction

```
require(
    state.config.dripcheck.check(state.config.checkparams),
    "Drippie: dripcheck failed so drip is not yet ready to be triggered"
);
```

Optimism provided some examples like checking if a `target` balance is below a specific threshold or above that threshold, but in general, the `dripcheck.check` invocation could perform any kind of checks.

The important part that should be clear in the natspec documentation of the `drip` function is that that specific check is performed only once before the execution of the bulk of actions.

Recommendation: Consider updating the natspec documentation of the `drip` function, making explicit that the check done by `dripcheck.check`is performed only before executing the batch of actions.

Optimism: The recommendation has been implemented in the PR#3280.

Spearbit: Fixed.

5.4.11 Considerations on the `drip` `state.last` and `state.config.interval` values

Severity: *Informational*

Context: Drippie.sol#L227-L230

Description: When the `drip` function is called by an external actor, the `executable` is executed to check if the `drip` meets all the needed requirements to be executed.

The only check that is done regarding the `drip` `state.last` and `state.config.interval` is this

```
require(
    state.last + state.config.interval <= block.timestamp,
    "Drippie: drip interval has not elapsed since last drip"
);
```

The `state.time` is never really initialized when the `create` function is called, this means that it will be automatically initialized with the default value of the `uint256` type: 0.

- Consideration 1: Drips could be executed as soon as created

Depending on the value set to `state.config.interval` the `executable`'s logic implies that as soon as a `drip` is created, the `drip` can be immediately (even in the same transaction) executed via the `drip` function.

- Consideration 2: A very high value for `interval` could make the drip never executable

`block.timestamp` represents the number of seconds that passed since `Unix Time` (1970-01-01T00:00:00Z). When the `owner` of the `Drippie` want to create a "one shot" drip that can be executed immediately after creation but only once (even if the owner forgets to set the drip's status to `ARCHIVED`) he/she should be aware that the max value that he/she can use for the `interval` is at max `block.timestamp`.

This mean that the second time the drip can be executed is after `block.timestamp` seconds have been passed. If, for example, the `owner` create right now a drip with `interval = block.timestamp` it means that after the first execution the same drip could be executed after ~52 years (~2022-1970).

Recommendation: At minimum, include documentation to highlight that drips could be immediately callable after creation, depending on the `interval` value. Consider the mentioned limitation of the `interval` max value if you want to have "one shot" actions that can be triggered as soon as created.

Consider adding an additional mechanism to manage the scenario where a drip should be executed *only after* a specific `timestamp` is passed.

5.4.12 Support ERC1155 in `AssetReceiver`

Severity: *Informational*

Context: AssetReceiver.sol#L128

Description: `AssetReceiver` support ERC20 and ERC721 interfaces but not ERC1155.

Recommendation: For generalized use cases, considering adding support for ERC1155.

5.4.13 Reorder `DripStatus` Enum for Clarity

Severity: *Informational*

Context: Drippie.sol#L30-L31

Description: The current implementation of `Drippie` contract has the following `enum` type:

```
enum DripStatus {
    NONE, // uint8(0)
    ACTIVE,
    PAUSED,
    ARCHIVED
}
```

When a `drip` is created via the `create` function, its status is initialized to `PAUSED` (equal to `uint8(2)`) and when it gets activated its status is changed to `ACTIVE` (equal to `uint8(1)`)

So, the status change from **0** (`NONE`) to **2** (`PAUSED`) to **1** (`ACTIVE`). Switching the order inside the `enum DripStatus` definition between `PAUSED` and `ACTIVE` would make it more clean and easier to understand.

Recommendation: Consider switching the order inside the `enum DripStatus` definition between `PAUSED` and `ACTIVE` would make it more clean and easier to understand.

Optimism: The recommendation has been implemented in the PR#3280.

Spearbit: Fixed.

5.4.14 `_gas` is Unneeded as `Transactor.CALL` and `Transactor.DELEGATECALL` Function Argument

Severity: *Informational*

Context: Transactor.sol#L30, Transactor.sol#L49

Description: The caller (i.e. contract `owner`) can control desired amount of gas at the transaction level.

Recommendation: Remove the `_gas` argument.

Optimism: Addressed in PR#3280.

Spearbit: Fixed.

5.4.15 Licensing Conflict on Inherited Dependencies

Severity: *Informational*

Context: Drippie.sol#L1, AssetReceiver.sol#L1, Transactor.sol#L1

Description: Solmate contracts are AGPL Licensed which is incompatible with the MIT License of Drippie related contracts.

Recommendation: Strict interpretations of AGPL require inheriting contracts to be released under AGPL.

Possible remediations include:

- Altering Drippie license
- Removing AGPL dependencies, using alternate library

Optimism: Spearbit v7 is now MIT licensed.

Spearbit: Solmate v7 license have been updated to MIT.

Note: The project has been audited with Solmate v6 (that has been audited) and not with Solmate v7 (which at the current time has not been audited).

5.4.16 Rename Functions for Clarity

Severity: *Informational*

Context: Drippie.sol#L160

Description:

`status` The `status(string memory _name, DripStatus _status)` function allows the owner to update the status of a `drip`.

The purpose of the function, based on the name, is not obvious at first sight and could confuse a user into believing that it's a *view* function to retrieve the status of a drip instead of mutating its status.

`executable`

The `executable(string memory _name) public view returns (bool)` function returns true if the drip with name `_name` can be executed.

Recommendation: Consider changing `status` to `setStatus`/`updateStatus`.

Consider changing `executable` to `isExecutable`.

5.4.17 Owner Has Permission to Drain Value from `Drippie` Contract

Severity: *Informational*

Context: Scenario 1: Drippie.sol#L126 Scenario 2: Drippie.sol#L19 Scenario 3: Transactor.sol#L27-L34

Description: Consider the following scenarios.

- Scenario 1

Owner may create arbitrary `drip`s, including a drip to send all funds to themselves.

- Scenario 2

`AssetReceiver` permits owner to withdraw ETH, ERC20 tokens, and ERC721 tokens.

- Scenario 3

Owner may execute arbitrary calls.

`Transactor.CALL` function is a function that allows the `owner` of the contract to execute a "general purpose" low-level call.

```
function CALL(
    address _target,
    bytes memory _data,
    uint256 _gas,
    uint256 _value
) external payable onlyOwner returns (bool, bytes memory) {
    return _target.call{ gas: _gas, value: _value }(_data);
}
```

The function will transfer `_value` ETH present in the contract balance to the `_target` address. The function is also `payable` and this mean that the owner can send along with the call some funds.

Test example showcasing the issue:

```
function test_transactorCALLAllowOwnerToDrainDrippieContract() public {
    bool success;

    vm.deal(deployer, 0 ether);
    vm.deal(bob, 0 ether);
    vm.deal(address(drippie), 1 ether);

    vm.prank(deployer);
    // send 1 ether via `call` to a contract that cannot receive them
    (success, ) = drippie.CALL{value: 0 ether}(bob, "", 100000, 1 ether);
    assertEq(success, true);

    assertEq(address(drippie).balance, 0 ether);
    assertEq(bob.balance, 1 ether);
}
```

Recommendation: These permissions appear intentional. Be sure to document for `Drippie` users and suggest they take any necessary precautions (multisigs, etc.).

Consider whether arbitrary `calls` are necessary and, if not, remove `AssetReceiver` (and inherit directly from `Owned`). Arbitrary calls can already be made by the owner creating a new drip, think of arbitrary calls as "one shot drips". Setting a very large `interval` makes it easy to `archive` a one shot drip after use.

What would be lost by removing `AssetReceiver` as a dependency is arbitrary state updates, from the `delegatecall`, and the `onERC721Received` we recommended adding to `AssetReceiver` in another ticket.