Drips
Security Review

Cantina Managed review by:
Optimum, Lead Security Researcher
Phaze, Security Researcher
Windhustler, Associate Security Researcher

July 5, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Drips is a protocol and app built on Ethereum that enables organizations and individuals to directly and publicly provide funding to the free and open source software projects they depend on the most.

Drips also includes gas-optimized and integrated primitives for streaming and splitting tokens, allowing users and web3 apps to stream and split funds by the second with continuous settlement for use cases like contributor payments, vesting and subscription memberships.

From Jun 4th to Jun 24th the Cantina team conducted a review of contracts on commit hash 5f601b7f. The team identified a total of **5** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 1
- Low Risk: 2
- Gas Optimizations: 0
- Informational: 1

3 Findings

3.1 High Risk

3.1.1 Non-executable messages in `BridgedGovernor` can result in an unrecoverable state

Severity: High Risk

Context: BridgedGovernor.sol#L57-L71

Description: If a non-executable message is relayed, `BridgedGovernor` may end up in an irreparable state due to the strict enforcement of sequential nonces.

The function used for receiving Layer Zero relayed messages (`BridgedGovernor.lzReceive`) strictly enforces sequential nonces. While this requirement is desired, it also introduces additional challenges and considerations if there are no alternative recovery mechanisms in place. If a relayed message is not executable on the receiving end, it is possible for `BridgedGovernor` to be unable to execute any further messages.

Messages can become undeliverable if they are malformed or if they contain non-executable sub calls. A simplified (and incomplete) example showcases the scenario:

```
pragma solidity ^0.8.20;

import {Test} from "forge-std/Test.sol";

import "src/BridgedGovernor.sol";

contract BridgedGovernorTest is Test {
    BridgedGovernor bridgedGovernor;
    address endpoint = address(bytes20(keccak256("endpoint")));

    uint32 ownerEid = 123;
    bytes32 owner = keccak256("owner");
    uint64 nonce = 1;

    function setUp() public {
        BridgedGovernor bridgedGovernorLogic = new BridgedGovernor(endpoint, ownerEid, owner);
        bridgedGovernor = BridgedGovernor(
            address(new BridgedGovernorProxy(address(bridgedGovernorLogic), new Call[](0)))
        );
    }

    function testMalformedCall() public {
        Origin memory origin = Origin(ownerEid, owner, nonce);
        bytes32 guid;
        // bytes memory message = abi.encode(new Call[](0)); // ok message
        bytes memory message = hex"deadbeef"; // malformed message
        address executor = address(this);
        bytes memory extraData;

        // this call cannot be executed
        vm.prank(endpoint);
        bridgedGovernor.lzReceive(origin, guid, message, executor, extraData);
    }
}
```

Since sequential messages are enforced and `BridgedGovernor` can only be upgraded through a call-to-self from a layer zero relayed message, it would not be able to upgrade its implementation to issue a fix.

Recommendation: Consider allowing the execution of messages with unordered nonces. An additional application level nonce can be introduced for additional guard rails when sending messages

```
  function nextNonce(uint32 srcEid, bytes32 sender)
      public
      view
      onlyProxy
      returns (uint64 nextNonce_)
  {
-     if (srcEid == ownerEid && sender == owner) nextNonce_ = _lastNonce + 1;
+     // enable unordered nonces on LZ
+     nextNonce_ = 0;
  }

  function lzReceive(
      Origin calldata origin,
      bytes32, /* guid */
      bytes calldata message,
      address, /* executor */
      bytes calldata /* extraData */
  ) public payable onlyProxy {
      require(msg.sender == endpoint, "Must be called by the endpoint");
      require(origin.srcEid == ownerEid, "Invalid message source chain");
      require(origin.sender == owner, "Invalid message sender");
-     require(origin.nonce == _lastNonce + 1, "Invalid message nonce");
+     // `appNonce` is encoded alongside the message
+     require(message.length >= 64, "Message too short");
+     (uint64 appNonce, Call[] memory calls) = abi.decode(message, (uint64, Call[]));
+     require(appNonce == _lastNonce + 1, "Invalid message nonce");
      // slither-disable-next-line events-maths
-     _lastNonce = origin.nonce;
-     runCalls(abi.decode(message, (Call[])));
+     _lastNonce = appNonce;
+     runCalls(calls);
  }
```

Drips: Fixed in commit 6a7b6c6a by implementing the auditor's recommendation.

Cantina Managed: Fixed.

3.2 Medium Risk

3.2.1 `BridgedGovernor.lzReceive` can be executed with different `msg.value` than intended

Severity: Medium Risk

Context: BridgedGovernor.sol#L57-L71

Description: `BridgedGovernor.lzReceive` logic doesn't check the address of the executor.

```
// BridgedGovernor.sol

function lzReceive(
    Origin calldata origin,
    bytes32, /* guid */
    bytes calldata message,
    address, /* executor */
    bytes calldata /* extraData */
) public payable onlyProxy {
    require(msg.sender == endpoint, "Must be called by the endpoint");
    require(origin.srcEid == ownerEid, "Invalid message source chain");
    require(origin.sender == owner, "Invalid message sender");
    require(origin.nonce == _lastNonce + 1, "Invalid message nonce");
    // slither-disable-next-line events-maths
    _lastNonce = origin.nonce;
    runCalls(abi.decode(message, (Call[])));
}
```

`EndpointV2.lzReceive` does not have access control and is freely callable by anyone once the message gets verified.

This means that a malicious actor can front-run the Executor and call `lzReceive` with a different `msg.value` than the sender has paid for. In certain scenarios depending on the encoded message data, this can result in a successful message being delivered but with a state update different than intended.

Impact: Medium/High as it can result in loss of tokens that were used to pay executor options on the sending chain and state update in one of the external calls different than intended.

Likelihood: Low as `msg.value` is checked for each external call.

Recommendation: Encode the `msg.value` inside the message on the sending chain, decode it in the `lzReceive` and compare it with the actual `msg.value`.

```
-      runCalls(abi.decode(message, (Call[])));
+     (uint256 msgValue, Call[] memory calls) = abi.decode(message, (uint256, Call[]));
+     require(msg.value >= msgValue, "Invalid message value");
+     runCalls(calls);
  }
```

Drips: Fixed in commit 6a7b6c6a by implementing the auditor's recommendation.

Cantina Managed: Fixed.

3.3 Low Risk

3.3.1 `WETH` implementation might not have the fallback function on various L2s leading to reverts

Severity: Low Risk

Context: Giver.sol#L137-L139

Description: `GiverRegistry` is initialized with the address of the `WETH` on the chain it's deployed on. It's done so if you're giving native tokens they are first wrapped in `WETH` and only then `WETH` get transferred. Wrapping into `WETH` is done by calling its fallback function.

WETH deployed on Ethereum has the following implementation:

```
// WETH.sol

function() public payable {
    deposit();
}

function deposit() public payable {
    balanceOf[msg.sender] += msg.value;
    Deposit(msg.sender, msg.value);
}
```

Calling the fallback function deposits native tokens into the `WETH` contract and mints the `WETH` tokens to the `msg.sender`. This is the intended behavior and fits within the giving logic.

As the team intends to deploy the `GiverRegistry` contract on L2s, it's important to confirm the `WETH` implementation is in line with the above-described behavior.

Moreover, the native token wrapper address should be passed as a constructor argument.

Recommendation: Pass the native token wrapper address as a constructor parameter and make sure the fallback function behavior is as expected.

Drips: Fixed in commit 0937f617 by implementing the auditor's recommendation.

Cantina Managed: Fixed.

3.3.2 `Giver` clones could unexpectedly call empty code

Severity: Low Risk

Context: Giver.sol#L95-L120

Description: A `Giver` clone may unexpectedly perform no operations in the case that the `GiversRegistry` is not initialized.

When `GiversRegistry.give` is called, an EIP-1167 minimal proxy clone is created. The clone's implementation contract points to a precomputed address which is the first contract deployed by `GiversRegistry`-the `Giver` contract. The `Giver` contract is deployed as a singleton contract when `GiversRegistry.initialize` is called. However, since the implementation contract address is computed deterministically, there is a possibility that the clone could point to an address that does not contain any code if `Giver` is not initialized.

The consequence of this is that calling `GiversRegistry.give` will result in the clone doing nothing, as it will perform a `delegatecall` into a contract address without any code. A call to an address without code will return `true` as the call's success status. This scenario could confuse users, as they might see a successful transaction without the desired effect of the clone transferring the tokens to the `AddressDriver` contract.

The impact is minimized because anyone can call `GiversRegistry.initialize`, which would deploy the `Giver` implementation contract at the correct address and enable the `Giver` clones to function properly.

Recommendation: Check the existence of contract and deploy `Giver` implementation if necessary or enforce an off-chain initialization check before `GiversRegistry.give` is called.

Drips: Fixed in commit 0d412126 by initializing a `Giver` contract inside the `give` function in case it does not exist.

Cantina Managed: Fixed.

3.4 Informational

3.4.1 ERC721 tokens are not recoverable from the `Giver` contract

Severity: Informational

Context: Giver.sol#L131-L150

Description: The `Giver` contract is located at a precomputed address based on an account ID. Users can send ERC20 tokens directly to this address, ensuring that the funds are correctly attributed and recoverable by the associated account ID.

However, if a user accidentally sends an ERC721 token to the `Giver` address, it may not be recoverable. Although ERC721 and ERC20 tokens have similar interfaces, calling `GiversRegistry.giveImpl` with an ERC721 token address and token ID instead of an ERC20 token address and amount is unlikely to pass the balance check located in `GiversRegistry.giveImpl`. Even if the call were to succeed, the ERC721 token would be sent to the `Drips` contract as a "Splits balance" as a result.

Recommendation: Consider implementing recovery paths for ERC721 tokens accidentally sent to the `Giver` contract. The `GiversRegistry` could include functionality to transfer ERC721 tokens to a DAO. Alternatively, the `GiversRegistry` could be upgraded to include this functionality if necessary.

Drips: Acknowledged, as of now won't fix.

Cantina Managed: Acknowledged.