Behodler contest
Findings & Analysis Report

2022-03-24

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 code 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 code contest outlined in this document, C4 conducted an analysis of a critical component of the Behodler Ecosystem. The code contest took place between January 27—February 2, 2022 with a specific focus on the Limbo smart contract system.

Wardens

39 Wardens contributed reports to the Behodler contest:

1. shw
2. kirk-baird
3. danb
4. sirhashalot
5. CertoraInc (danb, egjlmn1, OriDabush, ItayG, and shakedwinder)
6. Dravee
7. hyh
8. csanuragjain
9. camden
10. robee
11. Hawkeye (0xwags and 0xmint)
12. pauliax
13. cccz
14. Ruhum
15. wuwe1
16. Randyyy
17. jayjonah8

18. 0x1f8b
19. GeekyLumberjack
20. rfa
21. defsec
22. Fitraldys
23. BouSalman
24. gzeon
25. Tomio
26. ye0lde
27. throttle
28. 0v3rf10w
29. p4st13r4 (0x69e8 and 0xb4bb4)
30. bobi
31. cmichel
32. Jujic
33. lllllll

This contest was judged by Jack the Pug.

Final report assembled by liveactionllama, itsmetechjay, and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 33 unique vulnerabilities and 74 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 7 received a risk rating in the category of HIGH severity, 14 received a risk rating in the category of MEDIUM severity, and 12 received a risk rating in the category of LOW severity.

C4 analysis also identified 10 non-critical recommendations and 31 gas optimizations.

Scope

The code under review can be found within the C4 Behodler contest repository, and is composed of 14 smart contracts written in the Solidity programming language and includes 1629 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

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

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (7)

[H-01] Lack of access control on `assertGovernanceApproved` can cause funds to be locked

Submitted by shw, also found by kirk-baird and pauliax

Lack of access control on the `assertGovernanceApproved` function of `FlashGovernanceArbiter` allows anyone to lock other users' funds in the contract as long as the users have approved the contract to transfer `flashGovernanceConfig.amount` of `flashGovernanceConfig.asset` from them.

Proof of Concept

1. Alice wants to execute a flash governance decision (e.g., disable to the protocol), so she first calls `approve` on the `flashGovernanceConfig.asset` to allow `FlashGovernanceArbiter` to transfer `flashGovernanceConfig.amount` of assets from her.
2. An attacker Bob, who listens to the mempool, notices Alice's `approve` transaction and decides to front-run it. He calls `assertGovernanceApproved` with `sender` being Alice, `target` being any address, and `emergency` being `true`.
3. As a result, Alice cannot execute her flash governance decision, and her funds are locked in the contract for the `flashGovernanceConfig.unlockTime` period.

Referenced code: [DAO/FlashGovernanceArbiter.sol#L60-L81]

Recommended Mitigation Steps

Only allow certain addresses to call the `assertGovernanceApproved` function on `FlashGovernanceArbiter`.

gititGoro (Behodler) confirmed, but disagreed with High severity and commented:

> The reason I stuck with medium risk is because the user's funds can't be lost in this scenario. Only temporarily locked. If the user unapproves FlashGovernanceArbiter on EYE then they simply have to wait until the unlock period has passed and can withdraw again.

Jack the Pug (judge) commented:

> Agreed. This should be somewhere in between Med and High. If it's just the users' deposits being temporarily locked, then it's definitely a Med. But this one is taking probably all the funds from users' wallets and locking them against their will, easy to pull off by anyone, all at once for all potential victims.

> I tend to make it a High so that the future wardens and probably by extent the devs can be more careful with allowances. We have seen so many incidents caused by improper handling of users' allowances.

> A `transferFrom()` with from not being hard-coded as `msg.sender` is evil.

> My fellow wardens, if you are reading this, do not go easy on a `transferFrom()` that takes an argument as `from`.

[H-02] wrong minting amount

Submitted by danb

https://github.com/code-423n4/2022-01-behodler/blob/main/contracts/TokenProxies/RebaseProxy.sol#L36

```
    uint256 proxy = (baseBalance * ONE) / _redeemRate;
```

should be:

```
    uint256 proxy = (amount * ONE) / _redeemRate;
```

gititGoro (Behodler) confirmed, but disagreed with High severity and commented:

> Should be a balanceBefore and balanceAfter calculation with the diff being wrapped.

Jack the Pug (judge) commented:

> Valid `high`. The issue description can be more comprehensive though.

[H-03] Double transfer in the `transferAndCall` function of `ERC677`

Submitted by shw, also found by cccz, danb, and wuwe1

The implementation of the `transferAndCall` function in `ERC677` is incorrect. It transfers the `_value` amount of tokens twice instead of once. Since the `Flan`

contract inherits `ERC667`, anyone calling the `transferAndCall` function on `Flan` is affected by this double-transfer bug.

Proof of Concept

Below is the implementation of `transferAndCall`:

```
    function transferAndCall(
      address _to,
      uint256 _value,
      bytes memory _data
    ) public returns (bool success) {
      super.transfer(_to, _value);
      _transfer(msg.sender, _to, _value);
      if (isContract(_to)) {
          contractFallback(_to, _value, _data);
      }
      return true;
    }
```

We can see that `super.transfer(_to, _value);` and `_transfer(msg.sender, _to, _value);` are doing the same thing - transfering `_value` of tokens from `msg.sender` to `_to`.

Referenced code: ERC677/ERC677.sol#L28-L29

Recommended Mitigation Steps

Remove `_transfer(msg.sender, _to, _value);` in the `transferAndCall` function.

gititGoro (Behodler) confirmed and commented:

> Fix Behodler/limbo#3

[H-04] Logic error in `burnFlashGovernanceAsset` can cause locked assets to be stolen

Submitted by shw

A logic error in the `burnFlashGovernanceAsset` function that resets a user's `pendingFlashDecision` allows that user to steal other user's assets locked in future flash governance decisions. As a result, attackers can get their funds back even if they execute a malicious flash decision and the community burns their assets.

Proof of Concept

1. An attacker Alice executes a malicious flash governance decision, and her assets are locked in the `FlashGovernanceArbiter` contract.
2. The community disagrees with Alice's flash governance decision and calls `burnFlashGovernanceAsset` to burn her locked assets. However, the `burnFlashGovernanceAsset` function resets Alice's `pendingFlashDecision` to the default config (see line 134).
3. A benign user, Bob executes another flash governance decision, and his assets are locked in the contract.
4. Now, Alice calls `withdrawGovernanceAsset` to withdraw Bob's locked asset, effectively the same as stealing Bob's assets. Since Alice's `pendingFlashDecision` is reset to the default, the `unlockTime < block.timestamp` condition is fulfilled, and the withdrawal succeeds.

Referenced code: DAO/FlashGovernanceArbiter.sol#L134 DAO/FlashGovernanceArbiter.sol#L146

Recommended Mitigation Steps

Change line 134 to `delete pendingFlashDecision[targetContract][user]` instead of setting the `pendingFlashDecision` to the default.

gititGoro (Behodler) confirmed

[H-05] Flash loan price manipulation in `purchasePyroFlan()`

Submitted by sirhashalot

The comment on line 54 of FlanBackstop.sol states “the opportunity for price manipulation through flash loans exists”, and I agree that this is a serious risk. While the acceptableHighestPrice variable attempts to limit the maximum price change of the flan-stablecoin LP, a flashloan sandwich attack can still occur within this limit and make up for the limitation with larger volumes or multiple flashloan attacks. Flashloan price manipulation is the cause for many major hacks, including bZx, Harvest, and others.

Proof of Concept

Line 83 of FlanBackstop.sol calculates the price of flan to stablecoin in the Uniswap pool based on the balances at a single point in time. Pool balances at a single point in time can be manipulated with flash loans, which can skew the numbers to the extreme. The single data point of LP balances is used to calculate the growth variable in line 103, and the growth variable influences the quantity of pyroflan a user receives in the premium calculation on line 108.

```
uint256 priceBefore = (balanceOfFlanBefore * getMagnitude(stable
uint256 growth = ((priceBefore - tiltedPrice) * 100) / priceBefo
uint256 premium = (flanToMint * (growth / 2)) / 100;
```

Problems can occur when the volumes that the `purchasePyroFlan()` function sends to the Uniswap pool are large compared to the pool’s liquidity volume, or if the Uniswap pool price is temporarily tilted with a flashloan (or a whale). Because this function purposefully changes the exchange rate of the LP, by transferring tokens to the LP in a 2-to-1 ratio, a large volume could caught a large price impact in the LP. The code attempts to protect against this manipulation in line 102 with a require statement, but this can be worked around by reducing the volume per flashloan and repeating the attack multiple times. A user can manipulate the LP, especially when the LP is new with low liquidity, in order to achieve large amounts of flan and pyroflan.

Recommended Mitigation Steps

Use a TWAP instead of the pool price at a single point in time to increase the cost of performing a flashloan sandwich attack. See the Uniswap v2 price oracle solution documentation for more explanations on how Uniswap designed an approach to providing asset prices while reducing the change of manipulation.

gititGoro (Behodler) acknowledged and commented:

> This is a very well constructed report and if Flan was not intended to target a 1:1 with stablecoins, I'd accept it but since we know Flan shouldn't diverge far from 1:1, we don't run very large risks. Essentially, if the flan price crashes dramatically, backstop no longer works so the purpose of this contract is to just boost liquidity for Flan when Flan is operating under normal ish conditions. It's not intended to be black swan proof.

[H-06] Loss Of Flash Governance Tokens If They Are Not Withdrawn Before The Next Request

Submitted by kirk-baird

Users who have not called withdrawGovernanceAsset() after they have locked their tokens from a previous proposal (i.e. assertGovernanceApproved), will lose their tokens if assertGovernanceApproved() is called again with the same `target` and `sender`.

The `sender` will lose `pendingFlashDecision[target][sender].amount` tokens and the tokens will become unaccounted for and locked in the contract. Since the new amount is not added to the previous amount, instead the previous amount is overwritten with the new amount.

The impact of this is worsened by another vulnerability, that is assertGovernanceApproved() is a `public` function and may be called by any arbitrary user so long as the `sender` field has called `approve()` for `FlashGovernanceArbiter` on the ERC20 token. This would allow an attacker to make these tokens inaccessible for any arbitrary `sender`.

Proof of Concept

In assertGovernanceApproved() as seen below, the line `pendingFlashDecision[target][sender] = flashGovernanceConfig` will overwrite the previous contents. Thereby, making any previous rewards unaccounted for and inaccessible to anyone.

Note that we must wait `pendingFlashDecision[target][sender].unlockTime` between calls.

```
function assertGovernanceApproved(
  address sender,
  address target,
  bool emergency
) public {
  if (
    IERC20(flashGovernanceConfig.asset).transferFrom(sender, adc
    pendingFlashDecision[target][sender].unlockTime < block.time
  ) {
    require(
      emergency || (block.timestamp - security.lastFlashGovernar
      "Limbo: flash governance disabled for rest of epoch"
    );
    pendingFlashDecision[target][sender] = flashGovernanceConfic
    pendingFlashDecision[target][sender].unlockTime += block.tim

    security.lastFlashGovernanceAct = block.timestamp;
    emit flashDecision(sender, flashGovernanceConfig.asset, flas
  } else {
    revert("LIMBO: governance decision rejected.");
  }
}
```

Recommended Mitigation Steps

Consider updating the initial if statement to ensure the `pendingFlashDecision` for that `target` and `sender` is empty, that is:

```
function assertGovernanceApproved(
  address sender,
  address target,
  bool emergency
) public {
  if (
    IERC20(flashGovernanceConfig.asset).transferFrom(sender, adc
    pendingFlashDecision[target][sender].unlockTime == 0
  ) {
...
```

Note we cannot simply add the new `amount` to the previous `amount` incase the underlying `asset` has been changed.

gititGoro (Behodler) confirmed and commented:

> Excellent find! Thank you.

[H-07] LP pricing formula is vulnerable to flashloan manipulation

Submitted by shw

The LP pricing formula used in the `burnAsset` function of `LimboDAO` is vulnerable to flashloan manipulation. By swapping a large number of EYE into the underlying pool, an attacker can intentionally inflate the value of the LP tokens to get more `fate` than he is supposed to with a relatively low cost.

With the large portion of `fate` he gets, he has more voting power to influence the system's decisions, or even he can convert his `fate` to Flan tokens for a direct profit.

Proof of Concept

Below is an example of how the attack works:

1. Suppose that there are 1000 EYE and 1000 LINK tokens in the UniswapV2 LINK-EYE pool. The pool's total supply is 1000, and the attacker has 100 LP tokens.
2. If the attacker burns his LP tokens, he earns `1000 * 100/1000 * 20 = 2000` amount of `fate`.
3. Instead, the attacker swaps in 1000 EYE and gets 500 LINK from the pool (according to `x * y = k`, ignoring fees for simplicity). Now the pool contains 2000 EYE and 500 LINK tokens.
4. After the manipulation, he burns his LP tokens and gets `2000 * 100/1000 * 20 = 4000` amount of `fate`.

5. Lastly, he swaps 500 LINK into the pool to get back his 1000 EYE.
6. Compared to Step 2, the attacker earns a double amount of `fate` by only paying the swapping fees to the pool. The more EYE tokens he swaps into the pool, the more `fate` he can get. This attack is practically possible by leveraging flashloans or flashswaps from other pools containing EYE tokens.

The `setEYEBasedAssetStake` function has the same issue of using a manipulatable LP pricing formula. For more detailed explanations, please refer to the analysis of the Cheese Bank attack and the Warp Finance attack.

Referenced code: DAO/LimboDAO.sol#L356 DAO/LimboDAO.sol#L392

Recommended Mitigation Steps

Use a fair pricing formula for the LP tokens, for example, the one proposed by Alpha Finance.

gititGoro (Behodler) confirmed and commented:

> This is actually a good fate inflation vector especially when combined with the fateToFlan conversion

Jack the Pug (judge) commented:

> Good catch! A valid economic attack vector can potentially be exploited using flashloans.

Medium Risk Findings (14)

[M-01] Incorrect `unlockTime` can DOS `withdrawGovernanceAsset`

Submitted by csanuragjain

`unlockTime` is set incorrectly.

Proof of Concept

1. Navigate to contract at https://github.com/code-423n4/2022-01-behodler/blob/main/contracts/DAO/FlashGovernanceArbiter.sol
2. Observe the assertGovernanceApproved function

```
function assertGovernanceApproved(
    address sender,
    address target,
    bool emergency
  ) public {
...
pendingFlashDecision[target][sender].unlockTime += block.timesta
...
}
```

3. Assume assertGovernanceApproved is called with sender x and target y and pendingFlashDecision[target][sender].unlockTime is 100 and block.timestamp is 10000 then

```
pendingFlashDecision[target][sender].unlockTime += block.timesta
```

4. Again assertGovernanceApproved is called with same argument after timestamp 10100. This time unlockTime is set to very high value (assume block.timestamp is 10500). This is incorrect

```
pendingFlashDecision[target][sender].unlockTime += block.timesta
```

Recommended Mitigation Steps

Unlock time should be calculated like below:

```
constant public CONSTANT_UNLOCK_TIME = 1 days; // example
pendingFlashDecision[target][sender].unlockTime = CONSTANT_UNLOC
```

gititGoro (Behodler) confirmed and commented:

> Well spotted. This is a variant of a previously reported issue where the recommendation was to not allow flash governing a contract until stake has been withdrawn which is a safer fix.

Jack the Pug (judge) commented:

> I'm keeping this as `Med` instead of marking it as a duplicate of #156 as it did not illustrate the severe impact that it can cause.

[M-02] Reentrancy on Flash Governance Proposal Withdrawal

Submitted by kirk-baird

The function **withdrawGovernanceAsset()** is vulnerable to reentrancy, which would allow the attacker to drain the balance of the `flashGoveranance Config.asset`.

Note: this attack assumes the attacker may gain control of the execution flow in `asset.tranfer()` which is the case for many ERC20 tokens such as those that implement ERC777 but will depend on which asset is chosen in the configuration.

Proof of Concept

withdrawGovernanceAsset() does not follow the check-effects-interactions pattern as seen from the following code snippet, where an external call is made before state modifications.

```
function withdrawGovernanceAsset(address targetContract, address
  require(
    pendingFlashDecision[targetContract][msg.sender].asset == as
      pendingFlashDecision[targetContract][msg.sender].amount >
      pendingFlashDecision[targetContract][msg.sender].unlockTim
    "Limbo: Flashgovernance decision pending."
  );
  IERC20(pendingFlashDecision[targetContract][msg.sender].asset)
    msg.sender,
    pendingFlashDecision[targetContract][msg.sender].amount
  );
  delete pendingFlashDecision[targetContract][msg.sender];
```

```
}
```

The attacker can exploit this vulnerability through the following steps:

1. `assertGovernanceApproved(userA, target, false)`
2. wait for `unlockTime` seconds to pass
3. `withdrawGovernanceAsset(target, asset)` and gain control of the execution during `asset.transfer()`
4. repeat step 3) until there balance of `FlashGovernanceArbiter` is less than `pendingFlashDecision[target][msg.sender].amount`

Recommended Mitigation Steps

There are two possible mitigations, the first is to implement the check-effects-interactions patter. This involves doing as checks and state changes before making external calls. To implement this in the current context delete the `pendingFlashDecision` before making the external call as follows.

```
function withdrawGovernanceAsset(address targetContract, address
  require(
    pendingFlashDecision[targetContract][msg.sender].asset == as
      pendingFlashDecision[targetContract][msg.sender].amount >
      pendingFlashDecision[targetContract][msg.sender].unlockTim
    "Limbo: Flashgovernance decision pending."
  );
  uint256 amount = pendingFlashDecision[targetContract][msg.senc
  IERC20 asset = IERC20(pendingFlashDecision[targetContract][msg
  delete pendingFlashDecision[targetContract][msg.sender];
  asset.transfer(msg.sender, amount);
}
```

gititGoro (Behodler) acknowledged and commented:

> I do mention in the documentation that the only eligible assets are EYE and EYE LPs but that rule isn't enforced on a contract level which is why I'm acknowledging this rather than disputing it. Nonetheless it's not relevant to the context of Limbo

[M-03] Burning a User's Tokens for a Flash Proposal will not Deduct Their Balance

Submitted by kirk-baird

The proposal to burn a user's tokens for a flash governance proposal does not result in the user losing any funds and may in fact unlock their funds sooner.

Proof of Concept

The function burnFlashGovernanceAsset() will simply overwrite the user's state with `pendingFlashDecision[targetContract][user] = flashGovernanceConfig;` as seen below.

```
    function burnFlashGovernanceAsset(
      address targetContract,
      address user,
      address asset,
      uint256 amount
    ) public virtual onlySuccessfulProposal {
      if (pendingFlashDecision[targetContract][user].assetBurnable)
        Burnable(asset).burn(amount);
      }

      pendingFlashDecision[targetContract][user] = flashGovernanceCc
    }
```

Since `flashGovernanceConfig` is not modified in BurnFlashStakeDeposit.execute() the user will have `amount` set to the current config amount which is likely what they originally transferred in {assertGovernanceApproved()](https://github.com/code-423n4/2022-01-behodler/blob/main/contracts/DAO/FlashGovernanceArbiter.sol#L60).

Furthermore, `unlockTime` will be set to the config unlock time. The config unlock time is the length of time in seconds that proposal should lock tokens for not the future timestamp. That is unlock time may be say `7 days` rather than `now + 7 days`. As a result the check in withdrawGovernanceAsset() `pendingFlashDecision[targetContract][msg.sender].unlockTime < block.timestamp`, will always pass unless there is a significant misconfiguration.

Recommended Mitigation Steps

Consider deleting the user's data (i.e. `delete pendingFlashDecision[targetContract][user]`) rather than setting it to the config. This would ensure the user cannot withraw any funds afterwards.

Alternatively, only update `pendingFlashDecision[targetContract][user].amount` to subtract the amount sent as a function parameter and leave the remaining fields untouched.

gititGoro (Behodler) confirmed

[M-04] The system can get to a "stuck" state if a bad proposal (proposal that can't be executed) is accepted

Submitted by CertoraInc

LimboDAO.sol (`updateCurrentProposal()` modifier and `makeProposal()` function)

The LimboDAO contract has a variable that indicates the current proposal - every time there can be only one proposal. The only way a proposal can be done and a new proposal can be registered is to finish the previous proposal by either accepting it and executing it or by rejecting it. If a proposal that can't succeed, like for example an `UpdateMultipleSoulConfigProposal` proposal that has too much tokens and not enough gas, will stuck the system if it will be accepted. Thats because its time will pass - the users won't be able to vote anymore (because the `vote` function will revert), and the proposal can't be executed - the `execute` function will revert. So the proposal won't be able to be done and the system will be stuck because new proposal won't be able to be registered.

When trying to call the `executeCurrentProposal()` function that activates the `updateCurrentProposal()` modifier, the modifier will check the balance of fate, it will see that it's positive and will call `currentProposalState.proposal.orchestrateExecute()` to execute the proposal. the proposal will revert and cancel it all (leaving the proposal as the current proposal with `voting` state).

When trying to call `makeProposal()` function to make a new proposal it will revert because the current proposal is not equal to address(0).

To sum up, the system can get to a "stuck" state if a bad proposal (proposal that can't be executed) is accepted.

gititGoro (Behodler) confirmed and commented:

> I'm so glad someone finally noticed this. So many issues logged skirted around this issue. A lot of issues were logged about adding too may token to the updateMultipleSoulProposal but the crux of the matter is that the proposal.execute() should be replaced with a call that returns a success boolean so that the DAO doesn't get stuck on broken proposals. Congratulations on spotting this.

Jack the Pug (judge) decreased severity from High to Medium and commented:

> This is a good one, but I'm still going to downgrade this to `medium` as there is no fund at risk afaics.

gititGoro (Behodler) commented:

> @jack-the-pug There is a funds risk. Limbo can be paused via flash governance. When paused, funds can't be withdrawn. The only way to unpause is with a proposal. If the DAO gets jammed up with a broken proposal contract then an attacker can pause Limbo and all staked funds will be locked permanently.

Jack the Pug (judge) commented:

> Yeah, I agree that funds can be at risk indirectly, like the vector you described above, but only when the warden made a clear and persuasive presentation about how the funds can be at risk, then it can be a `High`.

> Furthermore, this attack vector requires the community to misbehave or at least be imprudent, to pass a malicious proposal, which already lowers the severity of it.

[M-05] flan can't be transferred unless the flan contract has flan balance greater than the amount we want to transfer

Submitted by CertoraInc

Flan.sol (`safeTransfer()` function)

The flan contract must have balance (and must have more flan then we want to transfer) in order to allow flan transfers. If it doesn't have any balance, the safeTransfer, which is the only way to transfer flan, will call `_transfer()` function with `amount = 0` . It should check `address(msg.sender)` 's balance instead of `address(this)` 's balance.

```
    function safeTransfer(address _to, uint256 _amount) external {
      uint256 flanBal = balanceOf(address(this)); // the problem is
      uint256 flanToTransfer = _amount > flanBal ? flanBal : _amount
      _transfer(_msgSender(), _to, flanToTransfer);
    }
```

gititGoro (Behodler) confirmed but disagreed with High severity

Jack the Pug (judge) decreased severity to Medium and commented:

> Downgrade to medium as there is no fund at risk.

[M-06] Consistently check account balance before and after transfers for Fee-On-Transfer discrepencies

Submitted by Dravee

Wrong fateBalance bookkeeping for a user. Wrong fateCreated value emitted.

Proof of Concept

Taking into account the FOT is done almost everywhere important in the solution already. That's a known practice in the solution.

However, it's missing here (see @audit-info tags):

```
File: LimboDAO.sol
383:   function burnAsset(address asset, uint256 amount) public
384:     require(assetApproved[asset], "LimboDAO: illegal asset'
385:     address sender = _msgSender();
386:     require(ERC677(asset).transferFrom(sender, address(this
387:     uint256 fateCreated = fateState[_msgSender()].fateBalar
388:     if (asset == domainConfig.eye) {
389:       fateCreated = amount * 10; //@audit-info wrong amount
390:       ERC677(domainConfig.eye).burn(amount);//@audit-info w
391:     } else {
392:       uint256 actualEyeBalance = IERC20(domainConfig.eye).b
393:       require(actualEyeBalance > 0, "LimboDAO: No EYE");
394:       uint256 totalSupply = IERC20(asset).totalSupply();
395:       uint256 eyePerUnit = (actualEyeBalance * ONE) / total
396:       uint256 impliedEye = (eyePerUnit * amount) / ONE;//@a
397:       fateCreated = impliedEye * 20;
398:     }
399:     fateState[_msgSender()].fateBalance += fateCreated; //@
400:     emit assetBurnt(_msgSender(), asset, fateCreated);//@au
401:   }
```

Tools Used

VS Code

Recommended Mitigation Steps

Check the balance before and after the transfer to take into account the Fees-On-Transfer.

gititGoro (Behodler) confirmed but disagreed with High severity and commented:

> Nice catch! It's not a level 3 bug, though.

Jack the Pug (judge) decreased severity to Medium and commented:

> Downgrade to `Med` as the assets need to be whitelisted.

[M-07] Calling `generateFLNQuote` twice in every block prevents any migration

Submitted by camden, also found by GeekyLumberjack, kirk-baird, and shw

https://github.com/code-423n4/2022-01-behodler/blob/71d8e0cfd9388f975d6a90dffba9b502b222bdfe/contracts/UniswapHelper.sol#L138 In the Uniswap helper, `generateFLNQuote` is public, so any user can generate the latest quote. If you call this twice in any block, then the two latest flan quotes will have a `blockProduced` value of the current block's number.

These quotes are used in the `_ensurePriceStability` function. The last require statement here is key: https://github.com/code-423n4/2022-01-behodler/blob/71d8e0cfd9388f975d6a90dffba9b502b222bdfe/contracts/UniswapHelper.sol#L283-L285

This function will revert if this statement is false:

```
localFlanQuotes[0].blockProduced - localFlanQuotes[1].blockProdu
```

Since `VARS.minQuoteWaitDuration` is a `uint256`, it is at least 0

```
localFlanQuotes[0].blockProduced - localFlanQuotes[1].blockProdu
```

But, as we've shown above, we can create a transaction in every block that will make `localFlanQuotes[0].blockProduced - localFlanQuotes[1].blockProduced == 0`. In any block we can make any call to `_ensurePriceStability` revert.

`_ensurePriceStability` is called in the `ensurePriceStability` modifier: https://github.com/code-423n4/2022-01-behodler/blob/71d8e0cfd9388f975d6a90dffba9b502b222bdfe/contracts/UniswapHelper.sol#L70

This modifier is used in `stabilizeFlan`: https://github.com/code-423n4/2022-01-

behodler/blob/71d8e0cfd9388f975d6a90dffba9b502b222bdfe/contracts/UniswapHelper.sol#L162

Lastly, `stabilizeFlan` is used in `migrate` in `Limbo.sol`
https://github.com/code-423n4/2022-01-behodler/blob/71d8e0cfd9388f975d6a90dffba9b502b222bdfe/contracts/Limbo.sol#L234

Therefore, we can grief a migration in any block. In reality, the `minQuoteWaitDuration` would be set to a much higher value than 0, making this even easier to grief for people (you only need to call `generateFLNQuote` every `minQuoteWaitDuration - 1` blocks to be safe).

Mitigation

Mitigation is to just use a time weighted oracle for uniswap.

gititGoro (Behodler) acknowledged and commented:

> I appreciate the write up. You're not technically incorrect on the problem. The solution isn't ideal because Uniswap can't tell us what's happening on Behodler.

> UniswapHelper can be replaced without much trouble. So if the oracle functionality does fail, we can deploy a better one. But for now, I doubt that the incentive exists to perpetually grief migrations on Limbo. If someone does start to grief, we can add flash governance to the flan quote generation and then burn the EYE belonging to griefers but I was reluctant to call on the big guns right from the start.

gititGoro (Behodler) changed to confirmed and commented:

> I've changed this to confirmed because a cryptoeconomic layer should be added. Flash governance for sampling an oracle is too extreme and adding a require to force the duration can still be griefed.

> Instead I think the solution is to force the caller to pay EYE if the interval is below the min required. The EYE is then burnt. So the idea is that if you wish to grief

> migrations, it's going to cost you more than just gas and the community will benefit from your griefing.

[M-08] Tolerance is not enforced during a flash governance decision

Submitted by shw

Most of the functions with a `governanceApproved` modifier call `flashGoverner.enforceTolerance` to ensure the provided parameters are restricted to some range of their original values. However, in the `governanceApproved` modifier, `flashGoverner.setEnforcement(true);` is called after the function body is executed, and thus the changed values are not restricted during the function execution.

An attacker can exploit this bug to change some critical parameters to arbitrary values by flash governance decisions. The effect will last until the community executes another proposal to correct the values. In the meanwhile, the attacker may make use of the corrupted values to launch an attack.

Proof of Concept

1. An attacker executes a flash governance decision, for example, the `adjustSoul` function of `Limbo`, and sets the `fps` of a soul to an extremely large value.
2. During the flash governance decision, some of his assets, for example, EYE, are locked in the `FlashGovernanceArbiter` contract.
3. He calls `claimReward` to get his rewards on the corresponding soul (assume that he has staked some number of the token before). Because of the manipulated `fps`, he gets a large number of Flan tokens as the reward.
4. Surely, he will lose his EYE tokens because of the malicious flash governance decision. However, as long as the attacker gets large enough Flan tokens, he is incentivized to launch such an attack.

Referenced code: DAO/Governable.sol#L46-L57 Limbo.sol#L380-L381 Limbo.sol#L327-L329 Limbo.sol#L530 Limbo.sol#L628-L630

Recommended Mitigation Steps

Rewrite the `_governanceApproved` function and the `governanceApproved` modifier as follows:

```
function _governanceApproved(bool emergency) internal {
  bool successfulProposal = LimboDAOLike(DAO).successfulProposal
  if (successfulProposal) {
    flashGoverner.setEnforcement(false);
  } else if (configured) {
    flashGoverner.setEnforcement(true);
    flashGoverner.assertGovernanceApproved(msg.sender, address(t
  }
}

modifier governanceApproved(bool emergency) {
  _governanceApproved(emergency);
  _;
}
```

gititGoro (Behodler) confirmed but disagreed with High severity and commented:

> So this is a vulnerability for the very first execution of flashgovernance decision on a contract, after which it's safe. This is the type of thing that won't be acted upon because it will have gone away by the time the public interacts with Limbo. However, it is technically true so I'm confirming the issue.

Jack the Pug (judge) decreased severity to Medium and commented:

> Valid finding, but the conditions are quite strict, downgraded to `Med`.

[M-09] All the scxMinted is at risk of being burnt.(Limbo.sol)

Submitted by Hawkeye

If one of the variables that calculate adjustedRectangle is a zero value,it will impair the calculation of excessSCX which would equal to all of the scxMinted on line 219.Nothing will be deducted from scxMinted on line 229 since adjustedRectangle =0 putting all of the former at risk of being burnt(line 230).

Also, the check on line 224 would not pass for high value migrations since scxMinted would always be greater than the adjustedRectangle.No scx would be avaliable to be sent to the AMM helper nor would there be any LP minted.

Furthermore, since SCX is needed to ensure the proper functioning of the protocol,ie, to provide liquidity and influence the value of Flan, it would be imperative that the correct value of excessScx is accounted for.

Recommended Mitigation Steps

Insert a require statement on line 222:

```
    require (AdjustedRectangle! =0, " err")
```

gititGoro (Behodler) acknowledged and commented:

> I really appreciate how deeply you've thought about this. Requires a thorough understanding of Limbo. Indeed your handle is apt (unless you're just naming yourself after the marvel hero in which case I would have to see your archery skills).

> The RectangleOfFairness is hardcoded as a constant 30 eth in Limbo.sol (line 269) so that can't be zero. The only way it could be zero is if the inflation factor is zero which is a community set variable. However, there might be some funny community edge case where they want it set to zero. For instance, suppose the community feels in some distant future that flan is sufficiently liquid but that SCX is still a bit dilute. Maybe they'd want to bring on new tokens while burning all new scx.

> I'm marking this as acknowledged, rather than disputed because your reasoning is really good.

[M-10] user won't be able to get his rewards in case of staking with amount = 0

Submitted by CertoraInc, also found by Randyyy

Limbo.sol (`stake()` function)

If a user has a pending reward and he calls the `stake` function with `amount = 0`, he won't be able to get his reward (he won't get the reward, and the reward debt will cover the reward)

That's happening because the reward calculation is done only if the staked amount (given as a parameter) is greater than 0, and it updates the reward debt also if the amount is 0, so the reward debt will be updated without the user will be able to get his reward

gititGoro (Behodler) confirmed and commented:

> Good catch! I'd be interested in your mitigation step being provided.

> To me, it looks like the simplest solution is just to remove that if statement. Users who stake zero will pay unnecessary gas costs but the contract shouldn't have to optimise gas consumption for undesired behaviour.

Jack the Pug (judge) increased severity from Low to Medium and commented:

> Upgraded to `Med` as users can lose their rewards.

[M-11] You can grief migrations by sending SCX to the UniswapHelper

Submitted by camden, also found by robee

The attack here allows the attacker to prevent migrations.

The attack here is recoverable because we can just call `buyFlanAndBurn` (f it worked as expected) with SCX as the input token to buy Flan with the extra SCX, then run the migration again.

Proof of Concept

The attack here is simple:

1. Get some SCX
2. Send it to the UniswapHelper contract
3. Any migration called will revert

My proof of concept test. You should be able to use this directly in the thig
https://gist.github.com/CamdenClark/b6841ac7a63e868d90eff7d9a40e3e0a

https://github.com/code-423n4/2022-01-behodler/blob/cedb81273f6daf2ee39ec765eef5ba74f21b2c6e/contracts/UniswapHelper.sol#L167

`localSCXBalance` is the SCX balance of the uniswap helper.
https://github.com/code-423n4/2022-01-behodler/blob/cedb81273f6daf2ee39ec765eef5ba74f21b2c6e/contracts/UniswapHelper.sol#L163

But, the caller of `stablizeFlan` assumes that the `rectangleOfFairness` parameter is going to be equal to the amount of SCX that was sent
https://github.com/code-423n4/2022-01-behodler/blob/cedb81273f6daf2ee39ec765eef5ba74f21b2c6e/contracts/Limbo.sol#L234

Recommended Mitigation Steps

The mitigation could be to do `>=` instead of `==` so sending tokens can't grief this.

Beyond this though, why do you need to pass in rectangleOfFairness if we're requiring it to be a function of the localSCXBalance anyways?
https://github.com/code-423n4/2022-01-behodler/blob/cedb81273f6daf2ee39ec765eef5ba74f21b2c6e/contracts/UniswapHelper.sol#L167

gititGoro (Behodler) acknowledged and commented:

> It's interesting to think about this issue because it's the type of tight rope walk between incentives and code enforcement. In this scenario, the net results of the griefing will be both a higher flan and scx price. So in exchange for a timely migration, we get a boost to flan and scx (which is precisely the goal of a

> migration from Limbo's perspective). Eventually we get the migration we wanted but only after some price assist from a griefer. For this cryptoeconomic reason, I've marked it as acknowledged rather than confirmed.

> Still I appreciate the depth with which you've thought about this and I hope the Behodler community sees more of you after this audit.

Jack the Pug (judge) commented:

> Good catch! Thank you for creating the proof of concept test script. The Code4rena community needs more wardens like you!

[M-12] You can flip governance decisions without extending vote duration

Submitted by camden, also found by kirk-baird

The impact here is that a user can, right at the end of the voting period, flip the decision without triggering the logic to extend the vote duration. The user doesn't even have to be very sophisticated: they can just send one vote in one transaction to go to 0, then in a subsequent transaction send enough to flip the vote.

Proof of Concept

https://github.com/code-423n4/2022-01-behodler/blob/608cec2e297867e4d954a63fecd720e80c1d5ae8/contracts/DAO/LimboDAO.sol#L281 You can send exactly enough fate to send the fate amount to 0, then send fate to change the vote. You'll never trigger this logic.

On the first call, to send the currentProposalState.fate to 0, `(fate + currentFate) * fate == 0`, so we won't extend the proposal state.

Then, on the second call, to actually change the vote, `fate * currentFate == 0` because `currentFate` is 0.

Recommended Mitigation Steps

Make sure that going to 0 is equivalent to a flip, but going away from 0 isn't a flip.

gititGoro (Behodler) confirmed and commented:

> Changing the logic to include this edge case can get a little convoluted. One thing I thought of is to change the condition to currentFate*fate<0 && currrentFate*currentFate>fate*fate but then moving from 0 to positive won't flip the vote. What about requiring the square of your vote to not equal the currentFate and reverting if not? In other words, your vote needs to either have no flipping impact or clearly be intended to flip, not just to cancel out all other votes.

gititGoro (Behodler) commented:

> After some consideration, I'm going to implement the square of votes != currentVote rule as a tie makes no sense in the context of whether to execute.

[M-13] Lack of access control in the `parameterize` function of proposal contracts

Submitted by shw, also found by hyh and jayjonah8

Most of the proposal contracts have a `parameterize` function for setting the proposal parameters, and these functions are protected only by the `notCurrent` modifier. When the proposal is proposed through a `lodgeProposal` transaction, an attacker can front-run it, modify the proposal parameters, and let the community vote it down. As a result, the person proposing loses his `fate` deposit.

Proof of Concept

1. A benign user Alice wants to make a proposal, so she deploys one of the proposal contracts and sets the intended parameters. Her proposal is approved by the `ProposalFactory` and is ready to be proposed.
2. Alice calls the `lodgeProposal` function of `ProposalFactory` to propose her proposal.
3. An attacker Bob, who listens to the mempool, notices Alice's transaction and front-runs it. He calls the `parameterize` function to change the parameters to undesirable ones.

4. Alice's proposal becomes the current proposal. However, the community rejects the proposal because of the changed parameters, causing Alice to lose her deposit.

Referenced code: DAO/Proposals/BurnFlashStakeDeposit.sol#L25-L37
DAO/Proposals/SetAssetApprovalProposal.sol#L21-L24
DAO/Proposals/ToggleWhitelistProposalProposal.sol#L22-L28
DAO/Proposals/UpdateMultipleSoulConfigProposal.sol#L40-L61
DAO/Proposals/WithdrawERC20Proposal.sol#L26-L32
DAO/ProposalFactory.sol#L74-L78

Recommended Mitigation Steps

Only allow the creator of the proposal to modify the parameters.

gititGoro (Behodler) acknowledged

[M-14] UniswapHelper.buyFlanAndBurn is a subject to sandwich attacks

Submitted by hyh

Trades can happen at a manipulated price and end up receiving fewer Flan to be bought than current market price dictates.

For example, at the time a user decides to call `buyFlanAndBurn` Flan trades at 0.8 in the input token terms at the corresponding DEX pool. If the input token holdings are big enough to compensate for pool manipulation costs, the following can happen: Flan buy order will be seen by a malicious bot, that buys Flan, pushing it to 0.9 before UniswapHelper's order comes through, and selling it back right afterwards. This way, given a cumulative impact of the trades on Flan's market price, the input token will be overspent.

This yields direct loss for the system as input token market operations have lesser effect than expected at the expense of contract holdings.

Proof of Concept

`buyFlanAndBurn` doesn't control for swap results, executing swaps with exchange pool provided amounts, which can be manipulated:

https://github.com/code-423n4/2022-01-behodler/blob/main/contracts/UniswapHelper.sol#L231

Recommended Mitigation Steps

Consider adding the minimum accepted price as a function argument so a user can limit the effective slippage, and check that actually received amount is above this accepted minimum.

Also, in the future it will prudent to add a relative version of the parameter to control percentage based slippage with TWAP Oracle price as a benchmark.

gititGoro (Behodler) acknowledged and commented:

> You're not wrong but remember that the tokens that can be called here are specifically those that are not listed on Limbo and likely never will be BUT that also have Flan pools. It's unlikely that these pools will ever be significantly large as no incentives are provided for their maintenance.

Low Risk Findings (12)

- **[L-01] `Limbo.sol` Does Not Implement `WithdrawERC20Proposal` Functionality** *Submitted by kirk-baird, also found by shw*
- **[L-02] Denial of Service in UpdateMultipleSoulConfigProposal** *Submitted by 0x1f8b, also found by Dravee, and robee*
- **[L-03] transfer return value of a general ERC20 is ignored** *Submitted by robee, also found by 0v3rf10w, 0x1f8b, bobi, BouSalman, cmichel, Dravee, Fitraldys, hyh, p4st13r4, Ruhum, and shw*
- **[L-04] Insufficient Validation of `burnFlashGovernanceAsset()` Parameters** *Submitted by kirk-baird*
- **[L-05] `approveUnstake` is unsafe** *Submitted by CertoraInc*
- **[L-06] Loss of precision in `purchasePyroFlan()`** *Submitted by sirhashalot, also found by Dravee*

- [L-07] Add emergency stop for specific stablecoins in `FlanBackstop` *Submitted by Ruhum, also found by hyh and robee*
- [L-08] `Governable` configuration can be backrun *Submitted by Ruhum, also found by kirk-baird*
- [L-09] `LimboDAO.killDAO()` doesn't update the DAO address of `FlanBackstop`, `UniswapHelper`, and `ProposalFactory` *Submitted by Ruhum*
- [L-10] Two Steps Verification before Transferring Ownership *Submitted by robee, also found by cccz*
- [L-11] Unstake wont work if pending reward is 0 *Submitted by csanuragjain, also found by CertoraInc and danb*
- [L-12] Proposal cost doesn't use votingDuration *Submitted by sirhashalot*

Non-Critical Findings (10)

- [N-01] UniswapHelper is open to manipulations on all chains whose id isn't 1 *Submitted by hyh, also found by 0x1f8b, Dravee, and pauliax*
- [N-02] Lack of Governance in Governable methods *Submitted by 0x1f8b, also found by wuwe1*
- [N-03] Wrong units in `convertFateToFlan()` *Submitted by sirhashalot*
- [N-04] require() validation and the revert message is not match *Submitted by rfa, also found by Dravee*
- [N-05] `LimboDAO.seed`: Wrong error message *Submitted by cmichel, also found by camden, danb, Dravee, and hyh*
- [N-06] Incorrect require statement *Submitted by csanuragjain*
- [N-07] Limbo, LimboDAO and FlashGovernanceArbiter events aren't indexed *Submitted by hyh, also found by p4st13r4*
- [N-08] commented debugging code *Submitted by BouSalman, also found by p4st13r4*
- [N-09] typo *Submitted by CertoraInc*
- [N-10] not emitting `ClaimedReward` event *Submitted by CertoraInc*

Gas Optimizations (31)

- [G-01] Caching array length can save gas *Submitted by robee, also found by BouSalman, CertoraInc, Dravee, gzeon, Jujic, pauliax, Randyyy, and throttle*
- [G-02] Gas in `FlashGovernanceArbiter.assertGovernanceApproved()` : `flashGovernanceConfig.asset` and `flashGovernanceConfig.amount` should get cached *Submitted by Dravee, also found by CertoraInc, hyh, and Ruhum*
- [G-03] Use calldata instead of memory *Submitted by robee, also found by Randyyy, rfa, sirhashalot, and Tomio*
- [G-04] Remove duplicate call to save gas *Submitted by Ruhum, also found by 0v3rf10w, camden, CertoraInc, Dravee, gzeon, hyh, and sirhashalot*
- [G-05] Revert string > 32 bytes *Submitted by sirhashalot, also found by BouSalman, Dravee, gzeon, pauliax, and robee*
- [G-06] inline a function (use its code) instead of calling it *Submitted by CertoraInc, also found by rfa, and robee*
- [G-07] Gas in `LimboDAO.seed()` : Avoiding a 2N for-loop for a N one *Submitted by Dravee*
- [G-08] Use of _msgSender() *Submitted by defsec*
- [G-09] Unnecessary default assignment *Submitted by robee, also found by BouSalman, Dravee, rfa, Ruhum, throttle, and yeOlde*
- [G-10] use multiple require() instead of && *Submitted by rfa*
- [G-11] Use != 0 instead of > 0 *Submitted by robee, also found by BouSalman, Dravee, gzeon, and Ruhum*
- [G-12] Prefix increments are cheaper than postfix increments *Submitted by robee, also found by 0x1f8b, CertoraInc, defsec, Dravee, lllllll, p4st13r4, rfa, and throttle*
- [G-13] Gas in `FlashGovernanceArbiter.enforceTolerance()` : substractions that can't underflow should be unchecked *Submitted by Dravee, also found by CertoraInc, defsec, pauliax, and sirhashalot*
- [G-14] a not needed variable *Submitted by CertoraInc*
- [G-15] Unnecessary if else in `UniswapHelper.configure()` *Submitted by Ruhum, also found by CertoraInc, sirhashalot, and yeOlde*
- [G-16] use a defined constant to save gas *Submitted by CertoraInc*

- [G-17] Immutable variables *Submitted by pauliax, also found by 0x1f8b, and robee*
- [G-18] Gas Optimization: Struct layout *Submitted by gzeon, also found by 0x1f8b, CertoraInc, Dravee, pauliax, and robee*
- [G-19] Gas saving removing variable *Submitted by 0x1f8b*
- [G-20] gas optimization by using shift operator *Submitted by BouSalman, also found by CertoraInc, and Dravee*
- [G-21] use variables indtead of array to save gas *Submitted by CertoraInc*
- [G-22] save gas by using `if else` instead of calculating the same expression twice *Submitted by CertoraInc*
- [G-23] Gas: "constants" expressions are expressions, not constants. *Submitted by Dravee*
- [G-24] Gas in `TransferHelper.ERC20NetTransfer` : check if amount != 0 before transfer *Submitted by Dravee, also found by csanuragjain and Tomio*
- [G-25] Gas in `UniswapHelper.configure()` : require statements should be reordered to save gas on revert *Submitted by Dravee*
- [G-26] dai already update on constructor *Submitted by Fitraldys*
- [G-27] Unnecessary constructor *Submitted by robee*
- [G-28] Using type(uint).max is cheaper than using calculation. *Submitted by Randyyy*
- [G-29] Gas savings *Submitted by csanuragjain*
- [G-30] Unused imports *Submitted by robee*
- [G-31] transferFrom gas improvement *Submitted by sirhashalot*

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