Inverse Finance contest Findings & Analysis Report

2022-12-20

Table of contents

Overview

About C4

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

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

During the audit contest outlined in this document, C4 conducted an analysis of the Inverse Finance smart contract system written in Solidity. The audit contest took place between October 25—October 30 2022.

Wardens

140 Wardens contributed reports to the Inverse Finance contest:

1. 0x1f8b
2. 0xNazgul
3. 0xRobocop
4. 0xRoxas
5. 0xSmartContract
6. 0xbepresent
7. 0xc0ffEE
8. 2997ms
9. 8olidity
10. Amithuddar
11. Aymen0909
12. B2
13. BClabs (nalus and Reptilia)
14. Bnke0x0
15. CertoraInc (egjlmn1, OriDabush, ItayG, shakedwinder, and RoiEvenHaim)
16. Ch_301
17. Chandr

18. Chom
19. CloudX (Migue, pabliyo, and marce1993)
20. Deivitto
21. Diana
22. Dinesh11G
23. ElKu
24. Franfran
25. HardlyCodeMan
26. Holmgren
27. JC
28. Jeiwan
29. Josiah
30. JrNet
31. Jujic
32. KoKo
33. Lambda
34. M4TZ1P (DekaiHako, holyhansss_kr, Zer0Luck, AAIIWITF, and exd0tpy)
35. Mathieu
36. MiloTruck
37. Olivierdem
38. Ozy42
39. Picodes
40. Rahoz
41. RaoulSchaffranek
42. RaymondFam
43. ReyAdmirado
44. Rolezn
45. Ruhum
46. Shinchan (Sm4rty, prasantgupta52, and Rohan16)

47. TomJ
48. Wawrdog
49. Waze
50. __141345__
51. adriro
52. ajtra
53. aphak5010
54. ballx
55. bin2chen
56. brgltd
57. c3phas
58. c7e7eff
59. carlitox477
60. catchup
61. cccz
62. cducrest
63. ch0bu
64. chaduke
65. chrisdior4
66. codexploder
67. corerouter
68. cryptonue
69. cryptostellar5
70. cryptphi
71. cuteboiz
72. cylzxje
73. d3e4
74. delfin454000
75. dipp

76. djxploit
77. durianSausage
78. eierina
79. elprofesor
80. enckrish
81. evmwanderer
82. exolorkistis
83. fatherOfBlocks
84. gogo
85. gs8nrv
86. hansfriese
87. horsefacts
88. idkwhatimdoing
89. imare
90. immeas
91. jayphbee
92. joestakey
93. jwood
94. kaden
95. karanctf
96. ladboy233
97. leosathya
98. lukris02
99. martin
100. mcwildy
101. minhtrng
102. neumo
103. oyc_109
104. pashov

105. peanuts
106. pedr02b2
107. pedroais
108. pfapostol
109. rbserver
110. ret2basic
111. robee
112. rokinot
113. rotcivegaf
114. rvierdiiev
115. sakman
116. sakshamguruji
117. sam_cunningham
118. saneryee
119. shark
120. simon135
121. skyle
122. sorrynotsorry
123. tnevler
124. tonisives
125. trustindistrust
126. wagmi
127. yamapyblack

This contest was judged by Oxean.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 18 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 18

received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 54 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 55 reports recommending gas optimizations.

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

Scope

The code under review can be found within the **C4 Inverse Finance contest repository**, and is composed of 8 smart contracts written in the Solidity programming language and includes 901 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/non-critical.

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**.

Medium Risk Findings (18)

[M-01] Unhandled return values of `transfer` and `transferFrom`

Submitted by 2997ms

ERC20 implementations are not always consistent. Some implementations of transfer and transferFrom could return 'false' on failure instead of reverting. It is safer to wrap such calls into `require()` statements to these failures.

Proof of Concept

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L205
https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L280
https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L399
https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L537
https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L570
https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L602

Recommended Mitigation Steps

Check the return value and revert on `0/false` or use OpenZeppelin's SafeERC20 wrapper functions.

08xmt (Inverse) acknowledged and commented:

> Every deployment of a market will use a trusted token, and be audited by the DAO and governance. Even when using safe transfer, there's no guarantee that an ERC20 token will behave as expected.

[M-02] Users can avoid paying fees if they manage to update their accrued fees periodically

Submitted by RaoulSchaffranek, *also found by* carlitox477

DBR.sol#L287

While a user borrows DOLA, his debt position in the DBR contract accrues more debt over time. However, Solidity contracts cannot update their storage

automatically over time; state updates must always be triggered by externally owned accounts. For this reason, the DBR contract cannot accurately represent a user's debt position in its storage at all times. Instead, the contract offers a method `accrueDueTokens` that, when called, updates the internal storage with the debts that accrued since the last update. This method is called before all critical financial operations that depend on an accurate value of the accumulated deficit in the contract's storage. On top, this method can also be invoked permissionless at any time. Suppose a borrower manages to call this function periodically and keep the time difference between updates short. In that case, a rounding error in the computation of the accrued debt can cause the expression to round down to zero. In this case, the user successfully avoided paying interest on his debt.

Proof of Concept

For reference, here is the affected code:

```
function accrueDueTokens(address user) public {
    uint debt = debts[user];
    if(lastUpdated[user] == block.timestamp) return;
    uint accrued = (block.timestamp - lastUpdated[user]) * c
    dueTokensAccrued[user] += accrued;
    totalDueTokensAccrued += accrued;
    lastUpdated[user] = block.timestamp;
    emit Transfer(user, address(0), accrued);
}
```

The problem is that the function updates the `lastUpdated[user]` storage variable even when `accrued` is `0`.

Example

Let's assume that the last update occurred at `t_0`.
Further assume that the next update occurs at `t_1` with `t_1 - t_0 = 12s`. (`12s` is the current Ethereum block time)
Suppose that the user's recorded `debt` position at `t_0` is `1,000,000 wei`.
Then the accrued debt formula gives us the following:

```
accrued = (t_1 - t_0) * debt / 365 days
```

```
= 12           * 1,000,000 / 31,536,000
= 1,000,000 / 31,536,000
= 0 (because unsigned integer division rounds down)
```

Maximizing profit

The accrued debt formula rounds towards zero if we have `(t_1 - t_0) * debt < 365 days`.
This gives us a method to compute the maximal debt that we can deposit to make the attack more efficient:

```
debt_max = 365 days / 12s -1 = 2,627,999
```

Notice that an attacker is not limited to these small loans. He can split a massive loan into multiple small loans, capped at 2,627,999.
To borrow X tokens (where X is given in WEI), we can compute the number of needed loans as:

```
#loans = X / 2,627,999
```

For example, to borrow 1 DOLA:

```
#loans = 10^18 / 2,627,999 = 380517648599
```

To borrow 1,000,000 DOLA we would thus need 380,517,648,599,000,000 small loans.

Economical feasibility

The attack would be economically feasible if the costs of the attack were lower than the interest that accrued throughout the successful attack.
The dominating factor of the attack costs is the gas costs which the attacker needs to pay to update the accrued interest of the small loans every second. A clever attacker would batch as many updates into a single transaction as possible to minimize the gas overhead of the transaction. Still, at the current block time (12s),

gas price (7 gwei), block gas limit (30,000,000), and current ETH price (`$1,550.80`), it's hardly imaginable that this attack is economically feasible at the moment.

Risk parameters

However, all these values could change in the future. And if we look at other networks, Layer2 or EVM compatible Layer1, the parameters might be different today.

Also, notice that if the contract were used to borrow a different asset than DOLA, the numbers would look drastically different. The risk increases with the asset's price and becomes bigger the fewer decimals the token uses. For example, to borrow 1 WBTC (8 decimals), we would only need 39 small loans:

```
#loans = 10^8 / 2,627,999 ~39
```

And to borrow WBTC worth $1,000,000 at a price of 20,746$/BTC, we would need 1864 small loans.

```
#loans ~= 49*10^8 / 2,627,999 ~= 1864
```

Foundry

The following test demonstrates how to avoid paying interest on a loan for 1h. A failing test means that the attack was successful.

```
$ git diff src/test/DBR.t.sol
diff --git a/src/test/DBR.t.sol b/src/test/DBR.t.sol
index 3988cf7..8779da7 100644
--- a/src/test/DBR.t.sol
+++ b/src/test/DBR.t.sol
@@ -25,6 +25,20 @@ contract DBRTest is FiRMTest {
         vm.stopPrank();
     }

+    function testFail_free_borrow() public {
+        uint borrowAmount =  2_627_999;
```

```
+
+         vm.prank(address(market));
+         dbr.onBorrow(user, borrowAmount);
+
+         for (uint i = 12; i <= 3600; i += 12) {
+             vm.warp(block.timestamp + 12);
+             dbr.accrueDueTokens(user);
+         }
+         assertEq(dbr.deficitOf(user), 0);
+     }
+
+
      function testOnBorrow_Reverts_When_AccrueDueTokensBringsUse
          gibWeth(user, wethTestAmount);
          gibDBR(user, wethTestAmount);
```

Output:

```
$ forge test --match-test testFail_free_borrow -vv
[⠒] Compiling...
[⠢] Compiling 1 files with 0.8.17
[⠆] Solc 0.8.17 finished in 2.62s
Compiler run successful

Running 1 test for src/test/DBR.t.sol:DBRTest
[FAIL. Reason: Assertion failed.] testFail_free_borrow() (gas: 1
Test result: FAILED. 0 passed; 1 failed; finished in 8.03ms

Failing tests:
Encountered 1 failing test in src/test/DBR.t.sol:DBRTest
[FAIL. Reason: Assertion failed.] testFail_free_borrow() (gas: 1

Encountered a total of 1 failing tests, 0 tests succeeded
```

Classified as a high medium because the yields can get stolen/denied. It's not high risk because I don't see an economically feasible exploit.

Tools Used

VSCode, Wolramapha, Foundry

Recommended Mitigation Steps

- Document the risks transparently and prominently.
- Re-evaluate the risks according to the specific network parameters of every network you want to deploy to.
- Do not update the `lastUpdated` timestamp of the user if the computed accrued amount was zero.

0xean (judge) commented:

> Debatable if this even qualifies as Medium. Leaning towards QA / LOW but will leave open for sponsor review.

08xmt (Inverse) confirmed and commented:

> Fixed in https://github.com/InverseFinance/FrontierV2/pull/20.

[M-03] User can borrow DOLA indefinitely without settling DBR deficit by keeping their debt close to the allowed maximum

Submitted by Holmgren

A user can borrow DOLA interest-free. This requires the user to precisely manage their collateral. This issue might become especially troublesome if a Market is opened with some stablecoin as the collateral (because price fluctuations would become negligible and carefully managing collateral level would be easy).

This issue is harder to exploit (but not impossible) if `gov` takes responsibility for forcing replenishment, since `gov` has a stronger economic incentive than third parties.

Proof of Concept

If my calculations are correct, with the current gas prices it costs about `$5` to call `Market.forceReplenish(...)`. Thus there is no economic incentive to do so as long as a debtor's DBR deficit is worth less than `$5` / `replenishmentIncentive` so probably around `$100`.

This is because replenishing cannot push a user's debt under the water (https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L567) and a user can repay their debt without having settled the DBR deficit (https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L531).

So, assuming the current prices, a user can:

1. Deposit some collateral
2. Borrow close to the maximum allowed amount of DOLA
3. Keep withdrawing or depositing collateral so that the collateral surplus does not exceed $100 (assuming current gas prices)
4. `repay()` their debt at any time in the future.
5. Withdraw all the collateral.

All this is possible with arbitrarily large DBR deficit because due to small collateral surplus at no point was it economical for a third party to `forceReplenish()` the user. If `gov` takes responsibility for `forceReplenish()` ing, the above procedure is still viable although the user has to maintain the collateral surplus at no more than around `$5`.

Recommended Mitigation Steps

Allow replenishing to push the debt under the water and disallow repaying the debt with an outstanding DBR deficit. E.g.:

```
diff --git a/src/Market.sol b/src/Market.sol
index 9585b85..d69b599 100644
--- a/src/Market.sol
+++ b/src/Market.sol
@@ -531,6 +531,7 @@ contract Market {
     function repay(address user, uint amount) public {
         uint debt = debts[user];
         require(debt >= amount, "Insufficient debt");
+        require(dbr.deficitOf(user) == 0, "DBR Deficit");
         debts[user] -= amount;
         totalDebt -= amount;
         dbr.onRepay(user, amount);
@@ -563,8 +564,6 @@ contract Market {
```

```
            uint replenishmentCost = amount * dbr.replenishmentPric
            uint replenisherReward = replenishmentCost * replenishm
            debts[user] += replenishmentCost;
-           uint collateralValue = getCollateralValueInternal(user)
-           require(collateralValue >= debts[user], "Exceeded colla
            totalDebt += replenishmentCost;
            dbr.onForceReplenish(user, amount);
            dola.transfer(msg.sender, replenisherReward);
```

0xean (judge) commented:

> This seems like a dust attack. Will leave open for sponsor review.

08xmt (Inverse) confirmed and commented:

> Fixed in https://github.com/InverseFinance/FrontierV2/pull/24.

[M-04] ERC777 reentrancy when withdrawing can be used to withdraw all collateral

Submitted by Lambda

Market.sol#L464

Markets can be deployed with arbitrary tokens for the collateral, including ERC777 tokens (that are downwards-compatible with ERC20). However, when the system is used with those tokens, an attacker can drain his escrow contract completely while still having a loan. This happens because with ERC777 tokens, there is a `tokensToSend` hook that is executed before the actual transfer (and the balance updates) happen. Therefore, `escrow.balance()` (which retrieves the token balance) will still report the old balance when an attacker reenters from this hook.

Proof Of Concept

We assume that `collateral` is an ERC777 token and that the `collateralFactorBps` is 5,000 (`50%`). The user has deposited 10,000 USD (worth of collateral) and taken out a loan worth 2,500 USD. He is therefore allowed to withdraw 5,000 USD (worth of collateral). However, he can usse the ERC777

reentrancy to take out all 10,000 USD (worth of collateral) and still keep the loaned 2,500 USD:

1. The user calls `withdraw(amount)` to withdraw his 5,000 USD (worth of collateral).
2. In `withdrawInternal`, the limit check succeeds (the user is allowed to withdraw 5,000 USD) and `escrow.pay(to, amount)` is called. This will initiate a transfer to the provided address (no matter which escrow is used, but we assume `SimpleERC20Escrow` for this example).
3. Because the collateral is an ERC777 token, the `tokensToSend` hook is executed before the actual transfer (and before any balance updates are made). The user can exploit this by calling `withdraw(amount)` again within the hook.
4. `withdrawInternal` will call `getWithdrawalLimitInternal`, which calls `escrow.balance()`. This receives the collateral balance of the escrow, which is not yet updated. Because of that, the balance is still 10,000 USD (worth of collateral) and the calculated withdraw limit is therefore still 5,000 USD.
5. Both transfers (the reentered one and the original one) succeed and the user has received all of his collateral (10,000 USD), while still having the 2,500 USD loan.

Recommended Mitigation Steps

Mark these functions as `nonReentrant`.

Oxean (judge) commented:

> Sponsor should review as the attack does seem valid with some pre-conditions (ERC777 tokens being used for collateral). Probably more of a Medium severity.

08xmt (Inverse) acknowledged, but disagreed with severity and commented:

> We make the security assumption that future collateral added by Inverse Finance DAO is compliant with standard ERC-20 behavior. Inverse Finance is full control of collateral that will be added to the platform and only intend to add collateral that properly reverts on failed transfers. Each ERC20 token added as collateral will be audited for non-standard behaviour. I would consider this a Low Risk finding, depending on how you value errors made in launch parameters.

0xean (judge) decreased severity to Medium and commented:

> @08xmt - The revert on a failed transfer here isn't the issue, it is the re-entrancy that isn't guarded against properly. While I understand your comment, if it were my codebase, I would simply add the modifier and incur the small gas costs as an additional layer of security to avoid mistakes in the future. I don't think this qualifies as High, but does show an attack path that *could* be achieved with an ERC777 token being used as collateral. Going to downgrade to Medium and will be happy to hear more discussion on the topic before final review.

08xmt (Inverse) commented:

> @0xean - The risk is still only present with unvetted contracts, and if the desire should exist in the future to implement a market with a token with re-entrancy, the code can be modified as necessary.

> Will respect the judge's decision on severity in the end, but ultimately seem like a deployment parameter risk more than anything.

0xean (judge) commented:

> Thanks @08xmt for the response.

> While I agree that proper vetting *could* avoid this issue, the wardens are analyzing the code and documentation that is presented before them and I think in light of this, the issue is valid. Had the warden simply stated that there was a reentrancy modifier missing without showing a valid path to it being exploited, I would downgrade to QA. But given they showed a valid attack path due to the lack of reentrancy controls I think this should be awarded.

[M-05] `repay` function can be DOSed

Submitted by djxploit, *also found by* immeas

Market.sol#L531

In `repay()` users can repay their debt.

```
function repay(address user, uint amount) public {
        uint debt = debts[user];
        require(debt >= amount, "Insufficient debt");
        debts[user] -= amount;
        totalDebt -= amount;
        dbr.onRepay(user, amount);
        dola.transferFrom(msg.sender, address(this), amount);
        emit Repay(user, msg.sender, amount);
    }
```

There is a `require` condition, that checks if the amount provided, is greater than the debt of the user. If it is, then the function reverts. This is where the vulnerability arises.

`repay` function can be frontrun by an attacker. Say an attacker pay a small amount of debt for the victim user, by frontrunning his repay transaction. Now when the victim's transaction gets executed, the `require` condition will fail, as the amount of debt is less than the amount of DOLA provided. Hence the attacker can repeat the process to DOS the victim from calling the repay function.

Proof of Concept

1. Victim calls repay() function to pay his debt of 500 DOLA , by providing the amount as 500
2. Now attacker saw this transaction on mempool
3. Attacker frontruns the transaction, by calling repay() with amount provided as 1 DOLA
4. Attacker's transaction get's executed first due to frontrunning, which reduces the debt of the victim user to 499 DOLA
5. Now when the victim's transaction get's executed, the debt of victim has reduced to 499 DOLA, and the amount to repay provided was 500 DOLA. Now as debt is less than the amount provided, so the require function will fail, and the victim's transaction will revert.

This will prevent the victim from calling repay function.

Hence an attacker can DOS the repay function for the victim user.

Recommended Mitigation Steps

Implement DOS protection.

0xean (judge) commented:

> This seems like a stretch to me. Will leave open for sponsor review but most likely close as invalid.

08xmt (Inverse) confirmed and commented:

> Mitigating PR: https://github.com/InverseFinance/FrontierV2/pull/13.

[M-06] User can free from liquidation fee if its escrow balance is less than the calculated liquidation fee

Submitted by jayphbee*, also found by* catchup*,* corerouter*,* trustindistrust*, and* cccz

User can free from liquidation fee if its escrow balance less than the calculated liquidation fee.

Proof of Concept

If the `liquidationFeeBps` is enabled, the `gov` should receive the liquidation fee. But if user's escrow balance is less than the calculated liquidation fee, `gov` got nothing.
https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L605-L610

```
            if(liquidationFeeBps > 0) {
                uint liquidationFee = repaidDebt * 1 ether / price *
                if(escrow.balance() >= liquidationFee) {
                    escrow.pay(gov, liquidationFee);
                }
            }
```

Recommended Mitigation Steps

User should pay all the remaining escrow balance if the calculated liquidation fee is greater than its escrow balance.

```
        if(liquidationFeeBps > 0) {
            uint liquidationFee = repaidDebt * 1 ether / price *
            if(escrow.balance() >= liquidationFee) {
                escrow.pay(gov, liquidationFee);
            } else {
                escrow.pay(gov, escrow.balance());
            }
        }
```

0xean (judge) commented:

> This should amount to dust.

08xmt (Inverse) confirmed and commented:

> Fixed in https://github.com/InverseFinance/FrontierV2/pull/15.

[M-07] Oracle's two-day feature can be gamed

Submitted by Ruhum

Oracle.sol#L124

The two-day feature of the oracle can be gamed where you only have to manipulate the oracle for ~2 blocks.

Proof of Concept

The oracle computes the day using:

```
    uint day = block.timestamp / 1 days;
```

Since we're working with `uint` values here, the following is true:

```
1728799 / 86400 = 1
172800 / 86400 = 2
```

Meaning, if you manipulate the oracle at the last block of day `X`, e.g. `23:59:50`, and at the first block of day `X + 1`, e.g. `00:00:02`, you bypass the two-day feature of the oracle. You only have to manipulate the oracle for two blocks.

This is quite hard to pull off. I'm also not sure whether there were any instances of Chainlink oracle manipulation before. But, since you designed this feature to prevent small timeframe oracle manipulation I think it's valid to point this out.

Recommended Mitigation Steps

If you increase it to a three-day interval you can fix this issue. Then, the oracle has to be manipulated for at least 24 hours.

08xmt (Inverse) acknowledged and commented:

> This is an issue if a 24 hour period elapses without any calls to the oracle *and* the underlying oracle is manipulable. The two day low is meant to be an added layer of security, but not bullet proof.

[M-08] Protocol withdrawals of collateral can be unexpectedly locked if governance sets the `collateralFactorBps` to 0

Submitted by trustindistrust, *also found by* cryptonue, d3e4, pashov, eierina, pedroais, RaoulSchaffranek, c7e7eff, simon135, Jujic, catchup, 0xbepresent, jwood, Lambda, peanuts, *and* codexploder

https://github.com/code-423n4/2022-10-inverse/blob/3e81f0f5908ea99b36e6ab72f13488bbfe622183/src/Market.sol#L359
https://github.com/code-423n4/2022-10-inverse/blob/3e81f0f5908ea99b36e6ab72f13488bbfe622183/src/Market.sol#L376

The FiRM `Marketplace` contract contains multiple governance functions for setting important values for a given debt market. Many of these are numeric values that affect ratios/levels for debt positions, fees, incentives, etc.

In particular, `Market.setCollateralFactorBps()` sets the ratio for how much collateral is required for loans vs the debt taken on by the user. The lower the value, the less debt a user can take on. See `Market.getCreditLimitInternal()` for that implementation.

The function `Market.getWithdrawalLimitInternal()` calculates how much collateral a user can withdraw from the protocol, factoring in their current level of debt. It contains the following check:

```
if(collateralFactorBps == 0) return 0;
```

This would cause the user to not be able to withdraw any tokens, so long as they had any non-0 amount of debt and the `collateralFactorBps` was 0.

Severity Rationalization

It is the warden's estimation that all semantics for locking functionality of the protocol should be explicit rather than implicit. While it is very unlikely that governance would intentionally set this value to 0, if it were to do so it would disproportionately affect users whose debt values were low compared to their deposited collateral.

It is also obvious that the same function that set the value to 0 could be used to revert the change. However, this would take time. Inverse Finance has mandatory minimums for the time required to process governance items in its workflow (https://docs.inverse.finance/inverse-finance/governance/creating-a-proposal)

> The community has a social agreement to post all proposals on the forum and as a draft in GovMills at least 24 hours before the proposal is put up for an on-chain vote, and also to host a community call focusing on the proposal before the voting period.

> Once a proposal has passed, it must be queued on-chain. This action can be triggered by anyone who is willing to pay the gas fee (usually done by a DAO

> member). The proposal then enters a holding period of 40 hours to allow users time to prepare for the consequences of the execution of the proposal.

As such, were the situation to occur it would cause at least 64 hours of lock.

Since the contract itself only overtly contains locking for new borrowing, this implicit lock on withdraws seems like an unnecessary risk.

Recommended Mitigation Steps

Consider a minimum for this value, to go along with the maximum value check already present in the setter function. While this will still reduce the quantity of collateral that can be withdrawn by users, it would allow for some withdraws to occur.

An explicit withdrawal lock could be implemented, making the semantic clear. This function could have modified access controls to enable faster reactions vs governance alone.

Alternatively, if there was an intention for this value to accept `0`, consider an 'escape hatch' function that could be enacted by users when a 'defaulted' state is set on the Market.

08xmt (Inverse) disputed and commented:

> This is functioning as intended. Setting a low collateralFactor like this is essentially a way to force borrowers to repay their debt. It may be a necessary operation in an emergency.

[M-09] Avoidable misconfiguration could lead to `INVEscrow` contract not minting `xINV` tokens

Submitted by **neumo**, *also found by* **minhtrng**, **ladboy233**, **BClabs**, *and* **rvierdiiev**

Market.sol#L281-L283

If a user creates a market with the **INVEscrow** implementation as **escrowImplementation** and false as **callOnDepositCallback**, the deposits made by

users in the escrow (through the market) would not mint **xINV** tokens for them. As **callOnDepositCallback** is an immutable variable set in the constructor, this mistake would make the market a failure and the user should deploy a new one (even worse, if the error is detected after any user has deposited funds, some sort of migration of funds should be needed).

Proof of Concept

Both **escrowImplementation** and **callOnDepositCallback** are immutable:

```
...
address public immutable escrowImplementation;
...
bool immutable callOnDepositCallback;
...
```

and its value is set at creation:

```
constructor (
        address _gov,
        address _lender,
        address _pauseGuardian,
        address _escrowImplementation,
        IDolaBorrowingRights _dbr,
        IERC20 _collateral,
        IOracle _oracle,
        uint _collateralFactorBps,
        uint _replenishmentIncentiveBps,
        uint _liquidationIncentiveBps,
        bool _callOnDepositCallback
    ) {
        ...
        escrowImplementation = _escrowImplementation;
        ...
        callOnDepositCallback = _callOnDepositCallback;
        ...
 }
```

When the user deposits collateral, if **callOnDepositCallback** is true, there is a call to the escrow's **onDeposit** callback:

```
function deposit(address user, uint amount) public {
        ...
        if(callOnDepositCallback) {
                escrow.onDeposit();
        }
        emit Deposit(user, amount);
}
```

This is **INVEscrow**'s onDeposit function:

```
function onDeposit() public {
        uint invBalance = token.balanceOf(address(this));
        if(invBalance > 0) {
                xINV.mint(invBalance); // we do not check returr
        }
}
```

The thing is if **callOnDepositCallback** is false, this function is never called and the user does not turn his/her collateral (**INV**) into **xINV**.

Recommended Mitigation Steps

Either make **callOnDepositCallback** a configurable parameter in Market.sol or always call the **onDeposit** callback (just get rid of the **callOnDepositCallback** variable) and leave it empty in case there's no extra functionality that needs to be executed for that escrow. In the case that the same escrow has to execute the callback for some markets and not for others, this solution would imply that there should be two escrows, one with the callback to be executed and another with the callback empty.

08xmt (Inverse) acknowledged, but disagreed with severity and commented:

> Fixed in https://github.com/InverseFinance/FrontierV2/pull/21/commits/0d4b01c594fb56a9f0ba944f6946874a5b335152

> We acknowledge that markets can be configured incorrectly, but it should generally be assumed that markets will be configured correctly, as this will go

> through both internal and governance review.

[M-10] Liquidation should make a borrower *healthier*

Submitted by hansfriese

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L559
https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L591

For a lending pool, borrower's debt healthness can be decided by the health factor, i.e. the collateral value divided by debt. (C/D)

The less the health factor is, the borrower's collateral is more risky of being liquidated.

Liquidation is supposed to make the borrower healthier (by paying debts and claiming some collateral), or else continuous liquidations can follow up and this can lead to a so-called liquidation crisis.

In a normal lending protocol, borrower's debt is limited by collateral factor in any case.

For this protocol, users can force replenishment for the addresses in deficit and the replenishment increases the borrower's debt.

And in the current implementation the replenishment is limited so that the new debt is not over than the collateral value.

As we will see below, this limitation is not enough and if the borrower's debt is over some threshold (still less than collateral value), liquidation makes the borrower debt "unhealthier".

And repeating liquidation can lead to various problems and we will even show an example that the attacker can take the DOLA out of the market.

Proof of Concept

Please see warden's original submission for full proof of concept.

Tools Used

Foundry

Recommended Mitigation Steps

Make sure the liquidation does not decrease the health index in the function `liquidate`.

With this mitigation, we also suggest limiting the debt increase in the function `forceReplenish` so that the new debt after replenish will not be over the threshold.

```
    function liquidate(address user, uint repaidDebt) public {
        require(repaidDebt > 0, "Must repay positive debt");
        uint debt = debts[user];
        require(getCreditLimitInternal(user) < debt, "User debt is h
        require(repaidDebt <= debt * liquidationFactorBps / 10000, '

        // ****************************************
        uint beforeHealthFactor = getCollateralValue(user) * 1e18 /
        // ****************************************

        uint price = oracle.getPrice(address(collateral), collateral
        uint liquidatorReward = repaidDebt * 1 ether / price; // col
        liquidatorReward += liquidatorReward * liquidationIncentiveI
        debts[user] -= repaidDebt;
        totalDebt -= repaidDebt;

        dbr.onRepay(user, repaidDebt);
        dola.transferFrom(msg.sender, address(this), repaidDebt);
        IEscrow escrow = predictEscrow(user);
        escrow.pay(msg.sender, liquidatorReward);
        if(liquidationFeeBps > 0) {
            uint liquidationFee = repaidDebt * 1 ether / price * lic
            if(escrow.balance() >= liquidationFee) {
                escrow.pay(gov, liquidationFee);
            }
        }

        // ****************************************
        uint afterHealthFactor = getCollateralValue(user) * 1e18 / c
        require(afterHealthFactor >= beforeHealthFactor, "Liquidatic
        // ****************************************

        emit Liquidate(user, msg.sender, repaidDebt, liquidatorRewar
```

```
    }

    function forceReplenish(address user, uint amount) public {
        uint deficit = dbr.deficitOf(user);
        require(deficit > 0, "No DBR deficit");
        require(deficit >= amount, "Amount > deficit");
        uint replenishmentCost = amount * dbr.replenishmentPriceBps(
        uint replenisherReward = replenishmentCost * replenishmentIr
        debts[user] += replenishmentCost;
        uint collateralValue = getCollateralValueInternal(user);

        // ****************************************
        // require(collateralValue >= debts[user], "Exceeded collate
        require(collateralValue >= debts[user] * (1 + liquidationInc
        // ****************************************

        totalDebt += replenishmentCost;
        dbr.onForceReplenish(user, amount);
        dola.transfer(msg.sender, replenisherReward);
        emit ForceReplenish(user, msg.sender, amount, replenishmentC
    }
```

08xmt (Inverse) confirmed and commented:

> Fixed by https://github.com/InverseFinance/FrontierV2/pull/22.

0xean (judge) decreased severity and commented:

> I think this comes down to design tradeoffs and is not unique to this specific lending protocol. It certainly shouldn't be consider High risk, but could see it being considered Medium as users should be aware that in market sell offs, cascading liquidations are a potential reality either due to liquidation rewards OR simply declining prices and the feedback loop at liquidations occur.

> That being said, these items are not unique to this protocol, so perhaps QA is a better grade for this issue.

0xean (judge) commented:

> @08xmt - care to weigh in on this one? I am unable to see your fix, but may help in how I judge it. The warden asked me to re-review and is suggesting a Medium

> severity.

08xmt (Inverse) commented:

> @0xean - I think a Medium rating is fair. Our fix has been to revert when the combination of Collateral Factor, Liquidation Incentive and Liquidation Fee would result in profitable self liquidations or unhealthier debt after liquidations.

```
if(collateralFactorBps > 0){
    uint unsafeLiquidationIncentive = 10000 * 10000 / cc
    require(liquidationIncentiveBps < unsafeLiquidation]
}
```

0xean (judge) increased severity to Medium and commented:

> Thanks, will upgrade back to Medium. :)

[M-11] `viewPrice` doesn't always report dampened price

Submitted by Jeiwan

Oracle.sol#L91

Oracle's `viewPrice` function doesn't report a dampened price until `getPrice` is called and today's price is updated. This will impact the public read-only functions that call it:

- getCollateralValue;
- getCreditLimit (calls `getCollateralValue`);
- getLiquidatableDebt (calls `getCreditLimit`);
- getWithdrawalLimit.

These functions are used to get on-chain state and prepare values for write calls (e.g. calculate withdrawal amount before withdrawing or calculate a user's debt that can be liquidated before liquidating it). Thus, wrong values returned by these functions can cause withdrawal of a wrong amount or liquidation of a wrong debt or cause reverts.

Proof of Concept

```
// src/test/Oracle.t.sol
function test_viewPriceNoDampenedPrice_AUDIT() public {
    uint collateralFactor = market.collateralFactorBps();
    uint day = block.timestamp / 1 days;
    uint feedPrice = ethFeed.latestAnswer();

    //1600e18 price saved as daily low
    oracle.getPrice(address(WETH), collateralFactor);
    assertEq(oracle.dailyLows(address(WETH), day), feedPrice);

    vm.warp(block.timestamp + 1 days);
    uint newPrice = 1200e18;
    ethFeed.changeAnswer(newPrice);
    //1200e18 price saved as daily low
    oracle.getPrice(address(WETH), collateralFactor);
    assertEq(oracle.dailyLows(address(WETH), ++day), newPrice);

    vm.warp(block.timestamp + 1 days);
    newPrice = 3000e18;
    ethFeed.changeAnswer(newPrice);

    //1200e18 should be twoDayLow, 3000e18 is current price. We
    // Notice that viewPrice is called before getPrice.
    uint viewPrice = oracle.viewPrice(address(WETH), collateralF
    uint price = oracle.getPrice(address(WETH), collateralFactor
    assertEq(oracle.dailyLows(address(WETH), ++day), newPrice);

    assertEq(price, 1200e18 * 10_000 / collateralFactor);

    // View price wasn't dampened.
    assertEq(viewPrice, 3000e18);
}
```

Recommended Mitigation Steps

Consider this change:

```
--- a/src/Oracle.sol
+++ b/src/Oracle.sol
@@ -89,6 +89,9 @@ contract Oracle {
```

```
                uint day = block.timestamp / 1 days;
                // get today's low
                uint todaysLow = dailyLows[token][day];
+               if(todaysLow == 0 || normalizedPrice < todaysLow) {
+                   todaysLow = normalizedPrice;
+               }
                // get yesterday's low
                uint yesterdaysLow = dailyLows[token][day - 1];
                // calculate new borrowing power based on collatera
```

0xean (judge) commented:

> Well written report that explains the impact of this unlike the others. Will leave open for review.

08xmt (Inverse) confirmed and commented:

> Fixed in https://github.com/InverseFinance/FrontierV2/pull/18.

[M-12] Users could get some `DOLA` even if they are on liquidation position

Submitted by Ch_301

Market.sol#L566

Users able to invoke `forceReplenish()` when they are on liquidation position.

Proof of Concept

On `Market.sol ==> forceReplenish()`
On this line

```
    uint collateralValue = getCollateralValueInternal(user);
```

`getCollateralValueInternal(user)` only return the value of the collateral

```
function getCollateralValueInternal(address user) internal r
    IEscrow escrow = predictEscrow(user);
    uint collateralBalance = escrow.balance();
    return collateralBalance * oracle.getPrice(address(colla
```

So if the user have 1.5 wETH at the price of `1 ETH = 1600 USD`
It will return `1.5 * 1600` and this value is the real value we can't just check it directly with the debt like this

```
require(collateralValue >= debts[user], "Exceeded collateral va
```

This is no longer `over collateralized` protocol.
The value needs to be multiplied by `collateralFactorBps / 10000`

- So depending on the value of `collateralFactorBps` and `liquidationFactorBps` the user could be in the liquidation position but he is able to invoke `forceReplenish()` to cover all their `dueTokensAccrued[user]` on `DBR.sol` and get more `DOLA`
- or it will lead a healthy debt to be in the liquidation position after invoking `forceReplenish()`
- *

Recommended Mitigation Steps

Use `getCreditLimitInternal()` rather than `getCollateralValueInternal()`.

Oxean (judge) commented:

> I believe this warden may be correct in the fact that we should actually be adding the `collateralFactor` into the check.

08xmt (Inverse) commented:

> While increasing debt beyond the Credit limit do risk creating bad debt, this bad debt is owed entirely to the protocol. If one wanted to minimise the amount of bad debt created this way, it would be possible to change the line to

> `getCollateralValueInternal() * (10000 - liquidationIncentiveBps) / 10000;` , as this would also slightly reduce the amount of bad debt paid out to force replenishers as incentives.

08xmt (Inverse) confirmed and commented:

> https://github.com/InverseFinance/FrontierV2/pull/17.

> Added a variant of this solution: https://github.com/code-423n4/2022-10-inverse-findings/issues/419#issuecomment-1313694712.

[M-13] `Market::forceReplenish` can be DoSed

Submitted by immeas

Market.sol#L562

If a user wants to completely `forceReplenish` a borrower with deficit, the borrower or any other malicious party can front run this with a dust amount to prevent the replenish.

Proof of Concept

```
function testForceReplenishFrontRun() public {
    gibWeth(user, wethTestAmount);
    gibDBR(user, wethTestAmount / 14);
    uint initialReplenisherDola = DOLA.balanceOf(replenisher

    vm.startPrank(user);
    deposit(wethTestAmount);
    uint borrowAmount = getMaxBorrowAmount(wethTestAmount);
    market.borrow(borrowAmount);
    uint initialUserDebt = market.debts(user);
    uint initialMarketDola = DOLA.balanceOf(address(market))
    vm.stopPrank();

    vm.warp(block.timestamp + 5 days);
    uint deficitBefore = dbr.deficitOf(user);
    vm.startPrank(replenisher);
```

```
        market.forceReplenish(user,1); // front run DoS

        vm.expectRevert("Amount > deficit");
        market.forceReplenish(user, deficitBefore); // fails due

        assertEq(DOLA.balanceOf(replenisher), initialReplenisher
        assertEq(DOLA.balanceOf(address(market)), initialMarketI
        assertEq(DOLA.balanceOf(replenisher) - initialReplenishe
            "DOLA balance of market did not decrease by amount p
        assertEq(dbr.deficitOf(user), deficitBefore-1, "Deficit

        // debt only increased by dust
        assertEq(market.debts(user) - initialUserDebt, 1 * reple
    }
```

This requires that the two txs end up in the same block. If they end up in different blocks the front run transaction will need to account for the increase in deficit between blocks.

Tools Used

vscode, forge

Recommended Mitigation Steps

Use `min(deficit,amount)` as amount to replenish.

Oxean (judge) commented:

> Very similar to #439 and unclear as the benefit the attacker is gaining here. They would be better off just front running the entire transaction and getting additional reward. Will leave open for sponsor review, but most likely QA or invalid.

08xmt (Inverse) confirmed and commented:

> Fixed in https://github.com/InverseFinance/FrontierV2/pull/16.
> Possible to imagine a situation where an attacker has an underwater loan and keeps front running his own forced replenishments with single digit DBR forced replenishments.

[M-14] Two day low oracle used in `Market.liquidate()` makes the system highly at risk in an oracle attack

Submitted by gs8nrv, *also found by* immeas, yamapyblack, idkwhatimdoing, kaden, Holmgren, *and* rvierdiiev

https://github.com/code-423n4/2022-10-inverse/blob/3e81f0f5908ea99b36e6ab72f13488bbfe622183/src/Market.sol#L596
https://github.com/code-423n4/2022-10-inverse/blob/3e81f0f5908ea99b36e6ab72f13488bbfe622183/src/Market.sol#L594
https://github.com/code-423n4/2022-10-inverse/blob/3e81f0f5908ea99b36e6ab72f13488bbfe622183/src/Market.sol#L597

Usage of the 2 day low exchange rate when trying to liquidate is highly risky as it incentives even more malicious agents to control the price feed for a short period of time. By controlling shortly the feed, it puts at risk any debt opened for a 2 day period + the collateral released will be overshoot during the liquidation.

Proof of Concept

The attack can be done by either an attack directly on the feed to push bad data, or in the case of Chainlink manipulating for a short period of time the markets to force an update from Chainlink. Then when either of the attacks has been made the attacker call `Oracle.getPrice()` . It then gives a 2 day period to the attacker (and any other agent who wants to liquidate) to liquidate any escrow.

This has a second drawback, we see that we use the same value at line 596, which is used to compute the liquidator reward (l.597), leading to more collateral released than expected. For instance manipulating once the feed and bring the ETH/USD rate to 20 instead of 2000, liquidator will earn 100 more than he should have had.

Recommended Mitigation Steps

Instead of using the 2 day lowest price during the liquidation, the team could either take the current oracle price, while still using the 2 day period for any direct agent interaction to minimise attacks both from users side and liquidators side.

0xean (judge) decreased severity to Medium

08xmt (Inverse) disputed and commented:

> The debt is not more at risk than through normal oracle manipulation. The oracle will return the normalized price if it's lower than the dampened two-day low, meaning oracle manipulations can always be used for bad liquidations.

[M-15] Oracle assumes token and feed decimals will be limited to 18 decimals

Submitted by adriro, *also found by* pashov, sorrynotsorry, neumo, Chom, CertoraInc, Ruhum, eierina, Lambda, RaoulSchaffranek, cryptphi, codexploder, BClabs, 8olidity, *and* joestakey

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Oracle.sol#L87
https://github.com/code-423n4/2022-10-inverse/blob/main/src/Oracle.sol#L121

The `Oracle` contract normalizes prices in both `viewPrices` and `getPrices` functions to adjust for potential decimal differences between feed and token decimals and the expected return value.

However these functions assume that `feedDecimals` and `tokenDecimals` won't exceed 18 since the normalization calculation is `36 - feedDecimals - tokenDecimals`, or that at worst case the sum of both won't exceed 36.

This assumption should be safe for certain cases, for example WETH is 18 decimals and the ETH/USD chainlink is 8 decimals, but may cause an overflow (and a revert) for the general case, rendering the Oracle useless in these cases.

Proof of Concept

If `feedDecimals + tokenDecimals > 36` then the expression `36 - feedDecimals - tokenDecimals` will be negative and (due to Solidity 0.8 default checked math) will cause a revert.

Recommended Mitigation Steps

In case `feedDecimals + tokenDecimals` exceeds 36, then the proper normalization procedure would be to **divide** the price by `10 ** decimals`. Something like this:

```
    uint normalizedPrice;

    if (feedDecimals + tokenDecimals > 36) {
        uint decimals = feedDecimals + tokenDecimals - 36;
        normalizedPrice = price / (10 ** decimals)
    } else {
        uint8 decimals = 36 - feedDecimals - tokenDecimals;
        normalizedPrice = price * (10 ** decimals);
    }
```

08xmt (Inverse) confirmed and commented:

> Fixed in https://github.com/InverseFinance/FrontierV2/pull/25
> Also pretty sure this is a dupe

[M-16] Calling `repay` function sends less DOLA to `Market` contract when `forceReplenish` function is not called while it could be called

Submitted by **rbserver**, *also found by* **Picodes**, **Ch_301**, **Jeiwan**, **ElKu**, **0xRobocop**, **MiloTruck**, *and* **sam_cunningham**

When a user incurs a DBR deficit, a replenisher can call the `forceReplenish` function to force the user to replenish DBR. However, there is no guarantee that the `forceReplenish` function will always be called. When the `forceReplenish` function is not called, such as because that the replenisher does not notice the user's DBR deficit promptly, the user can just call the `repay` function to repay the origianl debt and the `withdraw` function to receive all of the deposited collateral even when the user has a DBR deficit already. Yet, in the same situation, if the `forceReplenish` function has been called, more debt should be added for the user, and the user needs to repay more in order to get back all of the deposited collateral. Hence, when the `forceReplenish` function is not called while it could be called,

the `Market` contract would receive less DOLA if the user decides to repay the debt and withdraw the collateral both in full.

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L559-L572

```
    function forceReplenish(address user, uint amount) public {
        uint deficit = dbr.deficitOf(user);
        require(deficit > 0, "No DBR deficit");
        require(deficit >= amount, "Amount > deficit");
        uint replenishmentCost = amount * dbr.replenishmentPrice
        uint replenisherReward = replenishmentCost * replenishme
        debts[user] += replenishmentCost;
        uint collateralValue = getCollateralValueInternal(user);
        require(collateralValue >= debts[user], "Exceeded collat
        totalDebt += replenishmentCost;
        dbr.onForceReplenish(user, amount);
        dola.transfer(msg.sender, replenisherReward);
        emit ForceReplenish(user, msg.sender, amount, replenishm
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L531-L539

```
    function repay(address user, uint amount) public {
        uint debt = debts[user];
        require(debt >= amount, "Insufficient debt");
        debts[user] -= amount;
        totalDebt -= amount;
        dbr.onRepay(user, amount);
        dola.transferFrom(msg.sender, address(this), amount);
        emit Repay(user, msg.sender, amount);
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L472-L474

```
    function withdraw(uint amount) public {
        withdrawInternal(msg.sender, msg.sender, amount);
```

```
}
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L460-L466

```
function withdrawInternal(address from, address to, uint amo
    uint limit = getWithdrawalLimitInternal(from);
    require(limit >= amount, "Insufficient withdrawal limit'
    IEscrow escrow = getEscrow(from);
    escrow.pay(to, amount);
    emit Withdraw(from, to, amount);
}
```

Proof of Concept

Please add the following test in `src\test\Market.t.sol`. This test will pass to demonstrate the described scenario.

```
function testRepayAndWithdrawInFullWhenIncurringDBRDeficitIf
    gibWeth(user, wethTestAmount);
    gibDBR(user, wethTestAmount);

    vm.startPrank(user);

    // user deposits wethTestAmount WETH and borrows wethTes
    deposit(wethTestAmount);
    market.borrow(wethTestAmount);

    assertEq(DOLA.balanceOf(user), wethTestAmount);
    assertEq(WETH.balanceOf(user), 0);

    vm.warp(block.timestamp + 60 weeks);

    // after some time, user incurs DBR deficit
    assertGt(dbr.deficitOf(user), 0);

    // yet, since no one notices that user has a DBR deficit
    //   user is able to repay wethTestAmount DOLA that was
    market.repay(user, wethTestAmount);
    market.withdraw(wethTestAmount);
```

```
        vm.stopPrank();

        // as a result, user is able to get back all of the depo
        assertEq(DOLA.balanceOf(user), 0);
        assertEq(WETH.balanceOf(user), wethTestAmount);
    }
```

Tools Used

VSCode

Recommended Mitigation Steps

When calling the `repay` function, the user's DBR deficit can also be checked. If the user has a DBR deficit, an amount, which is similar to `replenishmentCost` that is calculated in the `forceReplenish` function, can be calculated; it can then be used to adjust the `repay` function's `amount` input for updating the states regarding the user's and total debts in the relevant contracts.

08xmt (Inverse) disputed and commented:

> Working as intended.

[M-17] Chainlink oracle data feed is not sufficiently validated and can return stale `price`

Submitted by rbserver, *also found by* d3e4, TomJ, pashov, sorrynotsorry, Aymen0909, c7e7eff, horsefacts, pedroais, minhtrng, dipp, 0xc0ffEE, Chom, immeas, imare, Olivierdem, Jeiwan, cccz, hansfriese, bin2chen, elprofesor, __141345__, tonisives, catchup, 0xNazgul, Rolezn, Ruhum, Franfran, Wawrdog, idkwhatimdoing, carlitox477, Lambda, peanuts, saneryee, djxploit, eierina, cuteboiz, martin, M4TZ1P, Jujic, rokinot, ladboy233, codexploder, 0x1f8b, joestakey, leosathya, rvierdiiev, *and* 8olidity

Calling the `Oracle` contract's `viewPrice` or `getPrice` function executes `uint price = feeds[token].feed.latestAnswer()` and `require(price > 0, "Invalid feed price")`. Besides that Chainlink's `latestAnswer` function is deprecated, only verifying that `price > 0` is true is also not enough to guarantee

that the returned `price` is not stale. Using a stale `price` can cause the calculations for the credit and withdrawal limits to be inaccurate, which, for example, can mistakenly consider a user's debt to be under water and unexpectedly allow the user's debt to be liquidated.

To avoid using a stale answer returned by the Chainlink oracle data feed, according to [Chainlink's documentation]:

1. The `latestRoundData` function can be used instead of the deprecated `latestAnswer` function.
2. `roundId` and `answeredInRound` are also returned. "You can check `answeredInRound` against the current `roundId`. If `answeredInRound` is less than `roundId`, the answer is being carried over. If `answeredInRound` is equal to `roundId`, then the answer is fresh."
3. "A read can revert if the caller is requesting the details of a round that was invalid or has not yet been answered. If you are deriving a round ID without having observed it before, the round might not be complete. To check the round, validate that the timestamp on that round is not 0."

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Oracle.sol#L78-L105

```
    function viewPrice(address token, uint collateralFactorBps)
        if(fixedPrices[token] > 0) return fixedPrices[token];
        if(feeds[token].feed != IChainlinkFeed(address(0))) {
            // get price from feed
            uint price = feeds[token].feed.latestAnswer();
            require(price > 0, "Invalid feed price");
            // normalize price
            uint8 feedDecimals = feeds[token].feed.decimals();
            uint8 tokenDecimals = feeds[token].tokenDecimals;
            uint8 decimals = 36 - feedDecimals - tokenDecimals;
            uint normalizedPrice = price * (10 ** decimals);
            uint day = block.timestamp / 1 days;
            // get today's low
            uint todaysLow = dailyLows[token][day];
            // get yesterday's low
            uint yesterdaysLow = dailyLows[token][day - 1];
            // calculate new borrowing power based on collateral
            uint newBorrowingPower = normalizedPrice * collatera
```

```
            uint twoDayLow = todaysLow > yesterdaysLow && yester
            if(twoDayLow > 0 && newBorrowingPower > twoDayLow) {
                uint dampenedPrice = twoDayLow * 10000 / collate
                return dampenedPrice < normalizedPrice ? dampene
            }
            return normalizedPrice;

        }
        revert("Price not found");
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Oracle.sol#L112-L144

```
    function getPrice(address token, uint collateralFactorBps) e
        if(fixedPrices[token] > 0) return fixedPrices[token];
        if(feeds[token].feed != IChainlinkFeed(address(0))) {
            // get price from feed
            uint price = feeds[token].feed.latestAnswer();
            require(price > 0, "Invalid feed price");
            // normalize price
            uint8 feedDecimals = feeds[token].feed.decimals();
            uint8 tokenDecimals = feeds[token].tokenDecimals;
            uint8 decimals = 36 - feedDecimals - tokenDecimals;
            uint normalizedPrice = price * (10 ** decimals);
            // potentially store price as today's low
            uint day = block.timestamp / 1 days;
            uint todaysLow = dailyLows[token][day];
            if(todaysLow == 0 || normalizedPrice < todaysLow) {
                dailyLows[token][day] = normalizedPrice;
                todaysLow = normalizedPrice;
                emit RecordDailyLow(token, normalizedPrice);
            }
            // get yesterday's low
            uint yesterdaysLow = dailyLows[token][day - 1];
            // calculate new borrowing power based on collateral
            uint newBorrowingPower = normalizedPrice * collatera
            uint twoDayLow = todaysLow > yesterdaysLow && yester
            if(twoDayLow > 0 && newBorrowingPower > twoDayLow) {
                uint dampenedPrice = twoDayLow * 10000 / collate
                return dampenedPrice < normalizedPrice ? dampene
            }
            return normalizedPrice;
```

```
        }
        revert("Price not found");
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L344-L347

```
    function getCreditLimitInternal(address user) internal retur
        uint collateralValue = getCollateralValueInternal(user);
        return collateralValue * collateralFactorBps / 10000;
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L323-L327

```
    function getCollateralValueInternal(address user) internal r
        IEscrow escrow = predictEscrow(user);
        uint collateralBalance = escrow.balance();
        return collateralBalance * oracle.getPrice(address(colla
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L353-L363

```
    function getWithdrawalLimitInternal(address user) internal r
        IEscrow escrow = predictEscrow(user);
        uint collateralBalance = escrow.balance();
        if(collateralBalance == 0) return 0;
        uint debt = debts[user];
        if(debt == 0) return collateralBalance;
        if(collateralFactorBps == 0) return 0;
        uint minimumCollateral = debt * 1 ether / oracle.getPric
        if(collateralBalance <= minimumCollateral) return 0;
        return collateralBalance - minimumCollateral;
    }
```

Proof of Concept

The following steps can occur for the described scenario.

1. Alice calls the `depositAndBorrow` function to deposit some WETH as the collateral and borrows some DOLA against the collateral.
2. Bob calls the `liquidate` function for trying to liquidate Alice's debt. Because the Chainlink oracle data feed returns an up-to-date price at this moment, the `getCreditLimitInternal` function calculates Alice's credit limit accurately, which does not cause Alice's debt to be under water. Hence, Bob's `liquidate` transaction reverts.
3. After some time, Bob calls the `liquidate` function again for trying to liquidate Alice's debt. This time, because the Chainlink oracle data feed returns a positive but stale price, the `getCreditLimitInternal` function calculates Alice's credit limit inaccurately, which mistakenly causes Alice's debt to be under water.
4. Bob's `liquidate` transaction is executed successfully so he gains some of Alice's WETH collateral. Alice loses such WETH collateral amount unexpectedly because her debt should not be considered as under water if the stale price was not used.

Tools Used

VSCode

Recommended Mitigation Steps

Oracle.sol#L82-L83 and Oracle.sol#L116-L117 can be updated to the following code.

```
(uint80 roundId, int256 answer, , uint256 updatedAt,
require(answeredInRound >= roundId, "answer is stale
require(updatedAt > 0, "round is incomplete");
require(answer > 0, "Invalid feed answer");

uint256 price = uint256(answer);
```

08xmt (Inverse) confirmed and commented:

> Fixed in https://github.com/InverseFinance/FrontierV2/pull/19

[M-18] Protocol's usability becomes very limited when access to Chainlink oracle data feed is blocked

Submitted by rbserver

Based on the current implementation, when the protocol wants to use Chainlink oracle data feed for getting a collateral token's price, the fixed price for the token should not be set. When the fixed price is not set for the token, calling the `Oracle` contract's `viewPrice` or `getPrice` function will execute `uint price = feeds[token].feed.latestAnswer()`. As https://blog.openzeppelin.com/secure-smart-contract-guidelines-the-dangers-of-price-oracles/ mentions, it is possible that Chainlink's "multisigs can immediately block access to price feeds at will". When this occurs, executing `feeds[token].feed.latestAnswer()` will revert so calling the `viewPrice` and `getPrice` functions also revert, which cause denial of service when calling functions like `getCollateralValueInternal` and `getWithdrawalLimitInternal`. The `getCollateralValueInternal` and `getWithdrawalLimitInternal` functions are the key elements to the core functionalities, such as borrowing, withdrawing, force-replenishing, and liquidating; with these functionalities facing DOS, the protocol's usability becomes very limited.

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Oracle.sol#L78-L105

```
function viewPrice(address token, uint collateralFactorBps)
    if(fixedPrices[token] > 0) return fixedPrices[token];
    if(feeds[token].feed != IChainlinkFeed(address(0))) {
        // get price from feed
        uint price = feeds[token].feed.latestAnswer();
        require(price > 0, "Invalid feed price");
        // normalize price
        uint8 feedDecimals = feeds[token].feed.decimals();
        uint8 tokenDecimals = feeds[token].tokenDecimals;
        uint8 decimals = 36 - feedDecimals - tokenDecimals;
        uint normalizedPrice = price * (10 ** decimals);
        uint day = block.timestamp / 1 days;
        // get today's low
        uint todaysLow = dailyLows[token][day];
        // get yesterday's low
        uint yesterdaysLow = dailyLows[token][day - 1];
```

```
            // calculate new borrowing power based on collateral
            uint newBorrowingPower = normalizedPrice * collatera
            uint twoDayLow = todaysLow > yesterdaysLow && yester
            if(twoDayLow > 0 && newBorrowingPower > twoDayLow) {
                uint dampenedPrice = twoDayLow * 10000 / collate
                return dampenedPrice < normalizedPrice ? dampene
            }
            return normalizedPrice;

        }
        revert("Price not found");
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Oracle.sol#L112-L144

```
    function getPrice(address token, uint collateralFactorBps) e
        if(fixedPrices[token] > 0) return fixedPrices[token];
        if(feeds[token].feed != IChainlinkFeed(address(0))) {
            // get price from feed
            uint price = feeds[token].feed.latestAnswer();
            require(price > 0, "Invalid feed price");
            // normalize price
            uint8 feedDecimals = feeds[token].feed.decimals();
            uint8 tokenDecimals = feeds[token].tokenDecimals;
            uint8 decimals = 36 - feedDecimals - tokenDecimals;
            uint normalizedPrice = price * (10 ** decimals);
            // potentially store price as today's low
            uint day = block.timestamp / 1 days;
            uint todaysLow = dailyLows[token][day];
            if(todaysLow == 0 || normalizedPrice < todaysLow) {
                dailyLows[token][day] = normalizedPrice;
                todaysLow = normalizedPrice;
                emit RecordDailyLow(token, normalizedPrice);
            }
            // get yesterday's low
            uint yesterdaysLow = dailyLows[token][day - 1];
            // calculate new borrowing power based on collateral
            uint newBorrowingPower = normalizedPrice * collatera
            uint twoDayLow = todaysLow > yesterdaysLow && yester
            if(twoDayLow > 0 && newBorrowingPower > twoDayLow) {
                uint dampenedPrice = twoDayLow * 10000 / collate
                return dampenedPrice < normalizedPrice ? dampene
            }
```

```
            return normalizedPrice;

        }
        revert("Price not found");
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L344-L347

```
    function getCreditLimitInternal(address user) internal retur
        uint collateralValue = getCollateralValueInternal(user);
        return collateralValue * collateralFactorBps / 10000;
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L323-L327

```
    function getCollateralValueInternal(address user) internal r
        IEscrow escrow = predictEscrow(user);
        uint collateralBalance = escrow.balance();
        return collateralBalance * oracle.getPrice(address(colla
    }
```

https://github.com/code-423n4/2022-10-inverse/blob/main/src/Market.sol#L353-L363

```
    function getWithdrawalLimitInternal(address user) internal r
        IEscrow escrow = predictEscrow(user);
        uint collateralBalance = escrow.balance();
        if(collateralBalance == 0) return 0;
        uint debt = debts[user];
        if(debt == 0) return collateralBalance;
        if(collateralFactorBps == 0) return 0;
        uint minimumCollateral = debt * 1 ether / oracle.getPric
        if(collateralBalance <= minimumCollateral) return 0;
        return collateralBalance - minimumCollateral;
    }
```

Proof of Concept

The following steps can occur for the described scenario.

1. Chainlink oracle data feed is used for getting the collateral token's price so the fixed price for the token is not set.
2. Alice calls the `depositAndBorrow` function to deposit some of the collateral token and borrows some DOLA against the collateral.
3. Chainlink's multisigs suddenly blocks access to price feeds so executing `feeds[token].feed.latestAnswer()` will revert.
4. Alice tries to borrow more DOLA but calling the `borrow` function, which eventually executes `feeds[token].feed.latestAnswer()`, reverts.
5. Alice tries to withdraw the deposited collateral but calling the `withdraw` function, which eventually executes `feeds[token].feed.latestAnswer()`, reverts.
6. Similarly, calling the `forceReplenish` and `liquidate` functions would all revert as well.

Tools Used

VSCode

Recommended Mitigation Steps

The `Oracle` contract's `viewPrice` and `getPrice` functions can be updated to refactor `feeds[token].feed.latestAnswer()` into `try feeds[token].feed.latestAnswer() returns (int256 price) { ... } catch Error(string memory) { ... }`. The logic for getting the collateral token's price from the Chainlink oracle data feed should be placed in the `try` block while some fallback logic when the access to the Chainlink oracle data feed is denied should be placed in the `catch` block. If getting the fixed price for the collateral token is considered as a fallback logic, then setting the fixed price for the token should become mandatory, which is different from the current implementation. Otherwise, fallback logic for getting the token's price from a fallback oracle is needed.

08xmt (Inverse) acknowledged, but disagreed with severity and commented:

> In the unlikely event of a chainlink msig block, the protocol can still recover through the use of governance actions to insert a new feed. I'd consider this a Low Severity, as protocol is only DOS'ed for a short period, and can't be repeatedly DOS'ed.

0xean (judge) commented:

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

> I don't think a Medium requires some amount of time for the DOS to be valid, so I think without a mitigation or fallback in place, this is a valid issue and should qualify as Medium.

08xmt (Inverse) commented:

> @0xean - That's fair.

Low Risk and Non-Critical Issues

For this contest, 54 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0x1f8b** received the top score from the judge.

The following wardens also submitted reports: JC, Deivitto, rbserver, d3e4, cylzxje, tnevler, c7e7eff, adriro, brgltd, horsefacts, c3phas, cryptonue, delfin454000, Aymen0909, Josiah, ReyAdmirado, rotcivegaf, cducrest, robee, gogo, lukris02, Waze, simon135, enckrish, wagmi, immeas, pedr02b2, sakshamguruji, hansfriese, ElKu, neumo, shark, __141345__, cryptostellar5, 0xSmartContract, 0xNazgul, trustindistrust, Rolezn, oyc_109, carlitox477, ch0bu, Diana, B2, evmwanderer, aphak5010, rvierdiiev, chrisdior4, Rahoz, Bnke0x0, Dinesh11G, fatherOfBlocks, RaymondFam, *and* leosathya.

[01] Allows malleable `SECP256K1` signatures

Here, the `ecrecover()` method doesn't check the `s` range.

Homestead (EIP-2) added this limitation, however the precompile remained unaltered. The majority of libraries, including OpenZeppelin, do this check.

Since an order can only be confirmed once and its hash is saved, there doesn't seem to be a serious danger in existing use cases.

Reference

- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/7201e6707f6631d9499a569f492870ebdd4133cf/contracts/utils/cryptography/ECDSA.sol#L138-L149

Affected Source Code

- DBR.sol:226-248
- Market.sol:425-447
- Market.sol:489-511

[02] Lack of checks `address(0)`

The following methods have a lack of checks if the received argument is an address, it's good practice in order to reduce human error to check that the address specified in the constructor or initialize is different than `address(0)`.

Affected Source Code

- BorrowController.sol:14
- BorrowController.sol:26
- SimpleERC20Escrow.sol:28
- GovTokenEscrow.sol:33-34
- INVEscrow.sol:35
- INVEscrow.sol:47-48
- Fed.sol:37-40
- Fed.sol:50
- Fed.sol:68
- Oracle.sol:32

- Oracle.sol:44
- DBR.sol:39
- DBR.sol:54
- Market.sol:77-83
- Market.sol:130
- Market.sol:136
- Market.sol:142

[03] Avoid using `tx.origin`

`tx.origin` is a global variable in Solidity that returns the address of the account that sent the transaction.

Using the variable could make a contract vulnerable if an authorized account calls a malicious contract. You can impersonate a user using a third party contract.

This can make it easier to create a vault on behalf of another user with an external administrator (by receiving it as an argument).

Affected Source Code

- BorrowController.sol:47

[04] Mixing and Outdated compiler

The pragma version used are:

```
pragma solidity ^0.8.13;
```

Note that mixing pragma is not recommended. Because different compiler versions have different meanings and behaviors, it also significantly raises maintenance costs. As a result, depending on the compiler version selected for any given file, deployed contracts may have security issues.

The minimum required version must be 0.8.17; otherwise, contracts will be affected by the following **important bug fixes**:

0.8.14:

- ABI Encoder: When ABI-encoding values from calldata that contain nested arrays, correctly validate the nested array length against `calldatasize()` in all cases.
- Override Checker: Allow changing data location for parameters only when overriding external functions.

0.8.15

- Code Generation: Avoid writing dirty bytes to storage when copying `bytes` arrays.
- Yul Optimizer: Keep all memory side-effects of inline assembly blocks.

0.8.16

- Code Generation: Fix data corruption that affected ABI-encoding of calldata values represented by tuples: structs at any nesting level; argument lists of external functions, events and errors; return value lists of external functions. The 32 leading bytes of the first dynamically-encoded value in the tuple would get zeroed when the last component contained a statically-encoded array.

0.8.17

- Yul Optimizer: Prevent the incorrect removal of storage writes before calls to Yul functions that conditionally terminate the external EVM call.

Apart from these, there are several minor bug fixes and improvements.

[05] Lack of ACK during owner change

It's possible to lose the ownership under specific circumstances.

Because of human error it's possible to set a new invalid owner. When you want to change the owner's address it's better to propose a new owner, and then accept this ownership with the new wallet.

Affected Source Code

- Fed.sol:50
- Market.sol:130

[06] Market pause is not checked during `contraction`

In the `Fed` contract, during the `expansion` method is checked that the market is not paused, this requirement is not done during the `contraction`.

```
    function contraction(IMarket market, uint amount) public {
        require(msg.sender == chair, "ONLY CHAIR");
        require(dbr.markets(address(market)), "UNSUPPORTED MARKE
+       require(!market.borrowPaused(), "CANNOT EXPAND PAUSED MA
        uint supply = supplies[market];
        require(amount <= supply, "AMOUNT TOO BIG"); // can't bu
        market.recall(amount);
        dola.burn(amount);
        supplies[market] -= amount;
        globalSupply -= amount;
        emit Contraction(market, amount);
    }
```

Affected Source Code

- Fed.sol:105

[07] Lack of no reentrant modifier

The `Market.getEscrow`, `Fed.expansion` and `Fed.contraction` methods do not have the `noReentrant` modifier and make calls to an external contract that can take advantage of and call these methods again, but it seems to fail due to the lack of tokens.

However, if any of the other addresses used their receive event to provide liquidity to the contract, the attacking account could benefit from it.

```
-   function expansion(IMarket market, uint amount) public {
+   function expansion(IMarket market, uint amount) public noRee
        ...
    }
```

```
-   function contraction(IMarket market, uint amount) public {
+   function contraction(IMarket market, uint amount) public noR
        ...
    }
```

For example, in `getEscrow` if the `escrow` allows a callback, it could create two scrows, loosing funds if in this callback it will call again `getEscrow`, using for example `deposit`

```
    function getEscrow(address user) internal returns (IEscrow)
        if(escrows[user] != IEscrow(address(0))) return escrows[
        IEscrow escrow = createEscrow(user);
        escrow.initialize(collateral, user);
        escrows[user] = escrow;
        return escrow;
    }
```

- Bob call `deposit`.
- During the `escrow` initialization it happend a reentrancy and call again `deposit`.
- The first deposit will be loss in the first escrow.

Please note that current escrows do not allow re-entry, so I decided to use Low. **It's always good to change the storage flags before the externals calls.**

Affected Source Code

- Fed.sol:86
- Fed.sol:103
- Market.sol:245

[08] Lack of checks the integer ranges

The following methods lack checks on the following integer arguments, you can see the recommendations above.

Affected Source Code

`_replenishmentPriceBps` is not checked to be != 0 during the `constructor`, nevertheless it's checked in [setReplenishmentPriceBps]

- [DBR.sol:36]

`replenishmentIncentiveBps` is not checked to be > 0 during the `constructor`, nevertheless it's checked in [setReplenismentIncentiveBps]

- [Market.sol:76]

[09] Lack of checks `supportsInterface`

The `EIP-165` standard helps detect that a smart contract implements the expected logic, prevents human error when configuring smart contract bindings, so it is recommended to check that the received argument is a contract and supports the expected interface.

Reference

- [https://eips.ethereum.org/EIPS/eip-165]

Affected Source Code

- [DBR.sol:99]
- [Market.sol:81-83]
- [Market.sol:118]
- [Market.sol:124]

[10] Lack of event emit

The `Market.pauseBorrows`, `Market.setLiquidationFeeBps`, `Market.setLiquidationIncentiveBps`, `Market.setReplenismentIncentiveBps`, `Market.setLiquidationFactorBps`, `Market.setCollateralFactorBps`, `Market.setBorrowController`, `Market.setOracle` methods do not emit an event when the state changes, something that it's very important for dApps and users.

Affected Source Code

- Market.sol:118
- Market.sol:124
- Market.sol:149
- Market.sol:161
- Market.sol:172
- Market.sol:183
- Market.sol:194
- Market.sol:218

[11] Oracle not compatible with tokens of 19 or more decimals

Keep in mind that the version of solidity used, despite being greater than `0.8`, does not prevent integer overflows during casting, it only does so in mathematical operations.

In the case that `feed.decimals()` returns 18, and the token is more than 18 decimals, the following subtraction will cause an underflow, denying the oracle service.

```
uint8 feedDecimals = feeds[token].feed.decimals();  // 18 =>
uint8 tokenDecimals = feeds[token].tokenDecimals;   // > 18
uint8 decimals = 36 - feedDecimals - tokenDecimals; // overf
```

All pairs have 8 decimals except the ETH pairs, so a token with 19 decimals in ETH, will fault.

Affected Source Code

- Oracle.sol:87-98

[12] Wrong visibility

The method `accrueDueTokens` doesn't check that the call is made by a market, and it's public, it should be changed to internal or private to be more resilient.

```
require(markets[msg.sender], "Only markets can call onBorrow");
```

Affected Source Code

- DBR.sol:284

[13] Bad nomenclature

The interface `IERC20` contains two methdos that are not pressent in the official ERC20, `delegate` and `delegates`, it's recommended to change the name of the contract because not any ERC20 it's valid.

Affected Source Code

- GovTokenEscrow.sol:9-10
- INVEscrow.sol:10-11

[14] Open TODO

The code that contains "open todos" reflects that the development is not finished and that the code can change a posteriori, prior release, with or without audit.

Affected Source Code

> // TODO: Test whether an immutable variable will persist across proxies

- INVEscrow.sol:35

[15] Avoid duplicate code

The `viewPrice` and `getPrice` methods of the `Oracle` contract are very similar, the only difference being the following peace of code:

```
if(todaysLow == 0 || normalizedPrice < todaysLow) {
    dailyLows[token][day] = normalizedPrice;
    todaysLow = normalizedPrice;
    emit RecordDailyLow(token, normalizedPrice);
}
```

It's recommended to reuse the code in order to be more readable and light.

Affected Source Code

- Oracle.sol:126-130
- Oracle.sol:79-103

[16] Avoid hardcoded values

It is not good practice to hardcode values, but if you are dealing with addresses much less, these can change between implementations, networks or projects, so it is convenient to remove these values from the source code.

Affected Source Code

- Market.sol:44

It's recommended to create a factor variable for `10000`:

- Market.sol:74-76
- Market.sol:150
- Market.sol:162
- Market.sol:173
- Market.sol:184
- Market.sol:195
- Market.sol:336
- Market.sol:346
- Market.sol:360
- Market.sol:377
- Market.sol:563-564
- Market.sol:583
- Market.sol:595-606

Gas Optimizations

For this contest, 55 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **pfapostol** received the top score from the judge.

The following wardens also submitted reports: mcwildy, sakman, JC, tnevler, ajtra, adriro, horsefacts, c3phas, Aymen0909, KoKo, ReyAdmirado, djxploit, robee, gogo, JrNet, 0xRoxas, enckrish, Amithuddar, CloudX, karanctf, Deivitto, Chandr, HardlyCodeMan, __141345__, shark, Shinchan, 0xSmartContract, sakshamguruji, Rolezn, ElKu, oyc_109, kaden, carlitox477, B2, ch0bu, martin, Ozy42, cryptostellar5, Diana, aphak5010, 0x1f8b, skyle, exolorkistis, durianSausage, Rahoz, Bnke0x0, ret2basic, Dinesh11G, ballx, fatherOfBlocks, chaduke, RaymondFam, Mathieu, *and* leosathya.

Summary

Gas savings are estimated using the gas report of existing `forge test --gas-report` tests (the sum of all deployment costs and the sum of the costs of calling methods) and may vary depending on the implementation of the fix.

	Issue	Instances	Estimated gas(deployments)	Estimated gas(min method call)	Estimated gas(avg method call)	Estimated gas(max method call)
01	State variables only set in the constructor should be declared immutable	2	117 275	104	110	110
02	Use function instead of modifiers	4	115 926	162	-264	-481
03	Duplicated `require()` / `revert()` checks should be refactored to a modifier or function	11	114 932	-59	-284	-398
04	Multiple address mappings can be combined into a single mapping of an address to a struct, where appropriate	5	24 227	254	533	-6 726
05	Expression can be unchecked when overflow is not possible	6	20 220	410	4 630	1354
06	State variables can be packed into fewer storage slots	1	-5 008	1 911	15 525	20 972

	Issue	Instances	Estimated gas(deployments)	Estimated gas(min method call)	Estimated gas(avg method call)	Estimated gas(max method call)
07	Refactoring similar statements	1	18 422	-18	-11	6
08	Better algorithm for underflow check	3	12 613	656	8 332	3 741
09	`x = x + y` is cheaper than `x += y`	12	11 214	180	468	616
10	`internal` functions only called once can be inlined to save gas	1	5 207	67	47	24
11	State variables should be cached in stack variables rather than re-reading them from storage	2	5 007	478	1 117	1 423
	Overall gas savings	**48**	**416 802 (6,58%)**	**3 423 (0,34%)**	**15 773 (0,82%)**	**18 283 (0,72%)**

Total: 48 instances over 11 issues

[01] State variables only set in the constructor should be declared immutable (2 instances)

Deployment. Gas Saved: **117 275**

Minimum Method Call. Gas Saved: **104**

Average Method Call. Gas Saved: **110**

Maximum Method Call. Gas Saved: **110**

Overall gas change: **-678 (-0.723%)**

Avoids a Gsset (20000 gas) in the constructor, and replaces each Gwarmacces (100 gas) with a PUSH32 (3 gas).

src/DBR.sol:11, 12

NOTE: name and symbol must be within 32 bytes

```
diff --git a/src/DBR.sol b/src/DBR.sol
index aab6daf..013960f 100644
--- a/src/DBR.sol
+++ b/src/DBR.sol
@@ -8,8 +8,8 @@ pragma solidity ^0.8.13;
     8,    8: */
     9,    9: contract DolaBorrowingRights {
    10,   10:
-   11      :-    string public name;
-   12      :-    string public symbol;
+       11:+    bytes32 public immutable name;
+       12:+    bytes32 public immutable symbol;
    13,   13:     uint8 public constant decimals = 18;
    14,   14:     uint256 public _totalSupply;
    15,   15:     address public operator;
@@ -34,8 +34,8 @@ contract DolaBorrowingRights {
    34,   34:         address _operator
    35,   35:     ) {
    36,   36:         replenishmentPriceBps = _replenishmentPriceB
-   37      :-        name = _name;
-   38      :-        symbol = _symbol;
+       37:+        name = bytes32(bytes(_name));
+       38:+        symbol = bytes32(bytes(_symbol));
    39,   39:         operator = _operator;
    40,   40:         INITIAL_CHAIN_ID = block.chainid;
    41,   41:         INITIAL_DOMAIN_SEPARATOR = computeDomainSepa
@@ -268,7 +268,7 @@ contract DolaBorrowingRights {
   268,  268:             keccak256(
   269,  269:                 abi.encode(
   270,  270:                     keccak256("EIP712Domain(string r
-  271     :-                     keccak256(bytes(name)),
+      271:+                     keccak256(bytes.concat(name)),
   272,  272:                     keccak256("1"),
   273,  273:                     block.chainid,
   274,  274:                     address(this)
```

[02] Use function instead of modifiers (4 instances)

Deployment. Gas Saved: **115 926**

Minimum Method Call. Gas Saved: **162**

Average Method Call. Gas Saved: **-264**

Maximum Method Call. Gas Saved: **-481**

Overall gas change: **734 (2.459%)**

src/BorrowController.sol:17

```
diff --git a/src/BorrowController.sol b/src/BorrowController.sol
index 6decad1..080a4e3 100644
--- a/src/BorrowController.sol
+++ b/src/BorrowController.sol
@@ -14,28 +14,36 @@ contract BorrowController {
    14,  14:         operator = _operator;
    15,  15:     }
    16,  16:
-   17     :-    modifier onlyOperator {
+        17:+    function onlyOperator() private view {
    18,  18:         require(msg.sender == operator, "Only operat
-   19     :-        _;
    20,  19:     }
    21,  20:
    22,  21:     /**
    23,  22:     @notice Sets the operator of the borrow controll
    24,  23:     @param _operator The address of the new operator
    25,  24:     */
-   26     :-    function setOperator(address _operator) public o
+        25:+    function setOperator(address _operator) public {
+        26:+        onlyOperator();
+        27:+        operator = _operator;
+        28:+    }
    27,  29:
    28,  30:     /**
    29,  31:     @notice Allows a contract to use the associated
    30,  32:     @param allowedContract The address of the allowe
    31,  33:     */
-   32     :-    function allow(address allowedContract) public o
+        34:+    function allow(address allowedContract) public {
+        35:+        onlyOperator();
+        36:+        contractAllowlist[allowedContract] = true;
+        37:+    }
    33,  38:
    34,  39:     /**
```

```
   35,  40:     @notice Denies a contract to use the associated
   36,  41:     @param deniedContract The addres of the denied c
   37,  42:     */
-  38     :-    function deny(address deniedContract) public onl
+       43:+    function deny(address deniedContract) public {
+       44:+        onlyOperator();
+       45:+        contractAllowlist[deniedContract] = false;
+       46:+    }
   39,  47:
   40,  48:     /**
   41,  49:     @notice Checks if a borrow is allowed
```

src/DBR.sol:44

```
diff --git a/src/DBR.sol b/src/DBR.sol
index aab6daf..50428cd 100644
--- a/src/DBR.sol
+++ b/src/DBR.sol
@@ -41,16 +41,16 @@ contract DolaBorrowingRights {
   41,  41:         INITIAL_DOMAIN_SEPARATOR = computeDomainSepa
   42,  42:     }
   43,  43:
-  44     :-    modifier onlyOperator {
+       44:+    function onlyOperator() private view {
   45,  45:         require(msg.sender == operator, "ONLY OPERAT
-  46     :-        _;
   47,  46:     }
   48,  47:
   49,  48:     /**
   50,  49:     @notice Sets pending operator of the contract. C
   51,  50:     @param newOperator_ The address of the newOperat
   52,  51:     */
-  53     :-    function setPendingOperator(address newOperator_
+       52:+    function setPendingOperator(address newOperator_
+       53:+        onlyOperator();
   54,  54:         pendingOperator = newOperator_;
   55,  55:     }
   56,  56:
@@ -59,7 +59,8 @@ contract DolaBorrowingRights {
   59,  59:      At 10000, the cost of replenishing 1 DBR is 1 I
   60,  60:     @param newReplenishmentPriceBps_ The new replen
   61,  61:     */
-  62     :-    function setReplenishmentPriceBps(uint newRepler
```

```
+       62:+    function setReplenishmentPriceBps(uint newReple
+       63:+        onlyOperator();
   63,  64:         require(newReplenishmentPriceBps_ > 0, "repl
   64,  65:         replenishmentPriceBps = newReplenishmentPric
   65,  66:     }
@@ -78,7 +79,8 @@ contract DolaBorrowingRights {
   78,  79:     @notice Add a minter to the set of addresses all
   79,  80:     @param minter_ The address of the new minter.
   80,  81:     */
-  81     :-    function addMinter(address minter_) public onlyO
+       82:+    function addMinter(address minter_) public {
+       83:+        onlyOperator();
   82,  84:         minters[minter_] = true;
   83,  85:         emit AddMinter(minter_);
   84,  86:     }
@@ -87,7 +89,8 @@ contract DolaBorrowingRights {
   87,  89:     @notice Removes a minter from the set of address
   88,  90:     @param minter_ The address to be removed from th
   89,  91:     */
-  90     :-    function removeMinter(address minter_) public or
+       92:+    function removeMinter(address minter_) public {
+       93:+        onlyOperator();
   91,  94:         minters[minter_] = false;
   92,  95:         emit RemoveMinter(minter_);
   93,  96:     }
@@ -96,7 +99,8 @@ contract DolaBorrowingRights {
   96,  99:     @dev markets can be added but cannot be removed.
   97, 100:     @param market_ The address of the new market cor
   98, 101:     */
-  99     :-    function addMarket(address market_) public onlyO
+      102:+    function addMarket(address market_) public {
+      103:+        onlyOperator();
  100, 104:         markets[market_] = true;
  101, 105:         emit AddMarket(market_);
  102, 106:     }
```

src/Market.sol:92

```
diff --git a/src/Market.sol b/src/Market.sol
index 9585b85..796d0d0 100644
--- a/src/Market.sol
+++ b/src/Market.sol
@@ -89,9 +89,8 @@ contract Market {
```

```
   89,  89:         INITIAL_DOMAIN_SEPARATOR = computeDomainSepa
   90,  90:     }
   91,  91:
-  92     :-    modifier onlyGov {
+       92:+    function onlyGov() private view {
   93,  93:         require(msg.sender == gov, "Only gov can cal
-  94     :-         _;
   95,  94:     }
   96,  95:
   97,  96:     function DOMAIN_SEPARATOR() public view virtual
@@ -115,38 +114,54 @@ contract Market {
  115, 114:     @notice sets the oracle to a new oracle. Only ca
  116, 115:     @param _oracle The new oracle conforming to the
  117, 116:     */
- 118     :-    function setOracle(IOracle _oracle) public onlyG
+       117:+    function setOracle(IOracle _oracle) public {
+       118:+        onlyGov();
+       119:+        oracle = _oracle;
+       120:+    }
  119, 121:
  120, 122:     /**
  121, 123:     @notice sets the borrow controller to a new borr
  122, 124:     @param _borrowController The new borrow controll
  123, 125:     */
- 124     :-    function setBorrowController(IBorrowController _
+       126:+    function setBorrowController(IBorrowController _
+       127:+        onlyGov();
+       128:+        borrowController = _borrowController;
+       129:+    }
  125, 130:
  126, 131:     /**
  127, 132:     @notice sets the address of governance. Only cal
  128, 133:     @param _gov Address of the new governance.
  129, 134:     */
- 130     :-    function setGov(address _gov) public onlyGov { g
+       135:+    function setGov(address _gov) public {
+       136:+        onlyGov();
+       137:+        gov = _gov;
+       138:+    }
  131, 139:
  132, 140:     /**
  133, 141:     @notice sets the lender to a new lender. The len
  134, 142:     @param _lender Address of the new lender.
  135, 143:     */
- 136     :-    function setLender(address _lender) public onlyG
+       144:+    function setLender(address _lender) public {
```

```
+      145:+        onlyGov();
+      146:+        lender = _lender;
+      147:+    }
  137, 148:
  138, 149:     /**
  139, 150:     @notice sets the pause guardian. The pause guarc
  140, 151:     @param _pauseGuardian Address of the new pauseGu
  141, 152:     */
- 142     :-    function setPauseGuardian(address _pauseGuardiar
+      153:+    function setPauseGuardian(address _pauseGuardiar
+      154:+        onlyGov();
+      155:+        pauseGuardian = _pauseGuardian;
+      156:+    }
  143, 157:
  144, 158:     /**
  145, 159:     @notice sets the Collateral Factor requirement c
  146, 160:     @dev Collateral factor mus be set below 100%
  147, 161:     @param _collateralFactorBps The new collateral f
  148, 162:     */
- 149     :-    function setCollateralFactorBps(uint _collateral
+      163:+    function setCollateralFactorBps(uint _collateral
+      164:+        onlyGov();
  150, 165:         require(_collateralFactorBps < 10000, "Inval
  151, 166:         collateralFactorBps = _collateralFactorBps;
  152, 167:     }
@@ -158,7 +173,8 @@ contract Market {
  158, 173:     @dev Must be set between 1 and 10000.
  159, 174:     @param _liquidationFactorBps The new liquidatior
  160, 175:     */
- 161     :-    function setLiquidationFactorBps(uint _liquidati
+      176:+    function setLiquidationFactorBps(uint _liquidati
+      177:+        onlyGov();
  162, 178:         require(_liquidationFactorBps > 0 && _liquic
  163, 179:         liquidationFactorBps = _liquidationFactorBps
  164, 180:     }
@@ -169,7 +185,8 @@ contract Market {
  169, 185:     @dev Must be set between 1 and 10000.
  170, 186:     @param _replenishmentIncentiveBps The new repler
  171, 187:     */
- 172     :-    function setReplenismentIncentiveBps(uint _reple
+      188:+    function setReplenismentIncentiveBps(uint _reple
+      189:+        onlyGov();
  173, 190:         require(_replenishmentIncentiveBps > 0 && _r
  174, 191:         replenishmentIncentiveBps = _replenishmentIr
  175, 192:     }
@@ -180,7 +197,8 @@ contract Market {
```

```
  180, 197:     @dev Must be set between 0 and 10000 - liquidati
  181, 198:     @param _liquidationIncentiveBps The new liqudati
  182, 199:     */
- 183     :-    function setLiquidationIncentiveBps(uint _liquic
+     200:+    function setLiquidationIncentiveBps(uint _liquic
+     201:+        onlyGov();
  184, 202:         require(_liquidationIncentiveBps > 0 && _lic
  185, 203:         liquidationIncentiveBps = _liquidationIncent
  186, 204:     }
@@ -191,7 +209,8 @@ contract Market {
  191, 209:     @dev Must be set between 0 and 10000 - liquidati
  192, 210:     @param _liquidationFeeBps The new liquidation fe
  193, 211:     */
- 194     :-    function setLiquidationFeeBps(uint _liquidationF
+     212:+    function setLiquidationFeeBps(uint _liquidationF
+     213:+        onlyGov();
  195, 214:         require(_liquidationFeeBps > 0 && _liquidati
  196, 215:         liquidationFeeBps = _liquidationFeeBps;
  197, 216:     }
```

src/Oracle.sol:35

```
diff --git a/src/Oracle.sol b/src/Oracle.sol
index 14338ed..3e7c608 100644
--- a/src/Oracle.sol
+++ b/src/Oracle.sol
@@ -32,16 +32,18 @@ contract Oracle {
   32,  32:         operator = _operator;
   33,  33:     }
   34,  34:
-  35     :-    modifier onlyOperator {
+     35:+    function onlyOperator() private view {
   36,  36:         require(msg.sender == operator, "ONLY OPERAT
-  37     :-        _;
   38,  37:     }
   39,  38:
   40,  39:     /**
   41,  40:     @notice Sets the pending operator of the oracle.
   42,  41:     @param newOperator_ The address of the pending o
   43,  42:     */
-  44     :-    function setPendingOperator(address newOperator_
+     43:+    function setPendingOperator(address newOperator_
+     44:+        onlyOperator();
```

```
+       45:+        pendingOperator = newOperator_;
+       46:+    }
    45,  47:
    46,  48:     /**
    47,  49:     @notice Sets the price feed of a specific token
@@ -50,7 +52,10 @@ contract Oracle {
    50,  52:     @param feed The chainlink feed of the ERC20 toke
    51,  53:     @param tokenDecimals uint8 representing the deci
    52,  54:     */
-   53     :-    function setFeed(address token, IChainlinkFeed f
+       55:+    function setFeed(address token, IChainlinkFeed f
+       56:+        onlyOperator();
+       57:+        feeds[token] = FeedData(feed, tokenDecimals)
+       58:+    }
    54,  59:
    55,  60:     /**
    56,  61:     @notice Sets a fixed price for a token
@@ -58,7 +63,10 @@ contract Oracle {
    58,  63:     @param token The address of the fixed price toke
    59,  64:     @param price The fixed price of the token. Remem
    60,  65:     */
-   61     :-    function setFixedPrice(address token, uint price
+       66:+    function setFixedPrice(address token, uint price
+       67:+        onlyOperator();
+       68:+        fixedPrices[token] = price;
+       69:+    }
    62,  70:
    63,  71:     /**
    64,  72:     @notice Claims the operator role. Only successfu
```

[03] Duplicated `require()` / `revert()` checks should be refactored to a modifier or function (instances)

Deployment. Gas Saved: **114 932**

Minimum Method Call. Gas Saved: **-59**

Average Method Call. Gas Saved: **-284**

Maximum Method Call. Gas Saved: **-398**

Overall gas change: **-2 665 (-12.599%)**

src/Fed.sol:49, 58, 67, 76, 87, 88, 104, 105

```
diff --git a/src/Fed.sol b/src/Fed.sol
index 1e819bb..8b54676 100644
--- a/src/Fed.sol
+++ b/src/Fed.sol
@@ -41,12 +41,24 @@ contract Fed {
    41,  41:         supplyCeiling = _supplyCeiling;
    42,  42:     }
    43,  43:
+       44:+    function is_gov() private view {
+       45:+        require(msg.sender == gov, "ONLY GOV");
+       46:+    }
+       47:+
+       48:+    function is_chair() private view {
+       49:+        require(msg.sender == chair, "ONLY CHAIR");
+       50:+    }
+       51:+
+       52:+    function is_supported_market(IMarket _market) pr
+       53:+        require(dbr.markets(address(_market)), "UNSU
+       54:+    }
+       55:+
    44,  56:     /**
    45,  57:     @notice Change the governance of the Fed contact
    46,  58:     @param _gov The address of the new governance cc
    47,  59:     */
    48,  60:     function changeGov(address _gov) public {
-   49     :-         require(msg.sender == gov, "ONLY GOV");
+       61:+        is_gov();
    50,  62:         gov = _gov;
    51,  63:     }
    52,  64:
@@ -55,7 +67,7 @@ contract Fed {
    55,  67:     @param _supplyCeiling Amount to set the supply c
    56,  68:     */
    57,  69:     function changeSupplyCeiling(uint _supplyCeiling
-   58     :-         require(msg.sender == gov, "ONLY GOV");
+       70:+        is_gov();
    59,  71:         supplyCeiling = _supplyCeiling;
    60,  72:     }
    61,  73:
@@ -64,7 +76,7 @@ contract Fed {
    64,  76:     @param _chair Address of the new chair.
    65,  77:     */
```

```
   66,  78:     function changeChair(address _chair) public {
-  67     :-         require(msg.sender == gov, "ONLY GOV");
+       79:+         is_gov();
   68,  80:         chair = _chair;
   69,  81:     }
   70,  82:
@@ -73,7 +85,7 @@ contract Fed {
   73,  85:     @dev Useful for immediately removing chair power
   74,  86:     */
   75,  87:     function resign() public {
-  76     :-         require(msg.sender == chair, "ONLY CHAIR");
+       88:+         is_chair();
   77,  89:         chair = address(0);
   78,  90:     }
   79,  91:
@@ -84,8 +96,8 @@ contract Fed {
   84,  96:     @param amount The amount of DOLA to mint and sup
   85,  97:     */
   86,  98:     function expansion(IMarket market, uint amount)
-  87     :-         require(msg.sender == chair, "ONLY CHAIR");
-  88     :-         require(dbr.markets(address(market)), "UNSUP
+       99:+         is_chair();
+      100:+         is_supported_market(market);
   89, 101:         require(market.borrowPaused() != true, "CANN
   90, 102:         dola.mint(address(market), amount);
   91, 103:         supplies[market] += amount;
@@ -101,8 +113,8 @@ contract Fed {
  101, 113:     @param amount The amount of DOLA to withdraw and
  102, 114:     */
  103, 115:     function contraction(IMarket market, uint amount
- 104     :-         require(msg.sender == chair, "ONLY CHAIR");
- 105     :-         require(dbr.markets(address(market)), "UNSUP
+      116:+         is_chair();
+      117:+         is_supported_market(market);
  106, 118:         uint supply = supplies[market];
  107, 119:         require(amount <= supply, "AMOUNT TOO BIG");
  108, 120:         market.recall(amount);
```

src/DBR.sol:171, 195, 373

```
diff --git a/src/DBR.sol b/src/DBR.sol
index aab6daf..625c422 100644
--- a/src/DBR.sol
```

```
+++ b/src/DBR.sol
@@ -46,6 +46,10 @@ contract DolaBorrowingRights {
    46,  46:         _;
    47,  47:     }
    48,  48:
+       49:+    function is_balance_sufficient(address _user, ui
+       50:+        require(balanceOf(_user) >= amount, "Insuffi
+       51:+    }
+       52:+
    49,  53:     /**
    50,  54:     @notice Sets pending operator of the contract. C
    51,  55:     @param newOperator_ The address of the newOperat
@@ -168,7 +172,7 @@ contract DolaBorrowingRights {
  168, 172:     @return Always returns true, will revert if not
  169, 173:     */
  170, 174:     function transfer(address to, uint256 amount) pu
- 171     :-         require(balanceOf(msg.sender) >= amount, "Ir
+     175:+         is_balance_sufficient(msg.sender, amount);
  172, 176:         balances[msg.sender] -= amount;
  173, 177:         unchecked {
  174, 178:             balances[to] += amount;
@@ -192,7 +196,7 @@ contract DolaBorrowingRights {
  192, 196:     ) public virtual returns (bool) {
  193, 197:         uint256 allowed = allowance[from][msg.sender
  194, 198:         if (allowed != type(uint256).max) allowance[
- 195     :-         require(balanceOf(from) >= amount, "Insuffic
+     199:+         is_balance_sufficient(from, amount);
  196, 200:         balances[from] -= amount;
  197, 201:         unchecked {
  198, 202:             balances[to] += amount;
@@ -370,7 +374,7 @@ contract DolaBorrowingRights {
  370, 374:     @param amount Amount of DBR to be burned.
  371, 375:     */
  372, 376:     function _burn(address from, uint256 amount) int
- 373     :-         require(balanceOf(from) >= amount, "Insuffic
+     377:+         is_balance_sufficient(from, amount);
  374, 378:         balances[from] -= amount;
  375, 379:         unchecked {
  376, 380:             _totalSupply -= amount;
```

[04] Multiple address mappings can be combined into a single mapping of an address to a struct, where appropriate (5 instances)

Deployment. Gas Saved: **24 227**

Minimum Method Call. Gas Saved: **254**

Average Method Call. Gas Saved: **533**

Maximum Method Call. Gas Saved: **-6 726**

Overall gas change: **-1 371 (20.741%)**

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (20000 gas) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

src/DBR.sol:19, 23, 26, 27, 28

```
diff --git a/src/DBR.sol b/src/DBR.sol
index aab6daf..43db0aa 100644
--- a/src/DBR.sol
+++ b/src/DBR.sol
@@ -8,6 +8,17 @@ pragma solidity ^0.8.13;
     8,    8: */
     9,    9: contract DolaBorrowingRights {
    10,   10:
+         11:+    struct UserInfo {
+         12:+        uint256 balances;
+         13:+
+         14:+        uint256 nonce;
+         15:+        uint256 debts;  // user => debt across all t
+         16:+        uint256 dueTokensAccrued; // user => amount
+         17:+        uint256 lastUpdated; // user => last update
+         18:+    }
+         19:+
+         20:+    mapping(address => mapping(address => uint256))
+         21:+
    11,   22:     string public name;
    12,   23:     string public symbol;
```

```
   13,  24:     uint8 public constant decimals = 18;
@@ -16,16 +27,11 @@ contract DolaBorrowingRights {
   16,  27:     address public pendingOperator;
   17,  28:     uint public totalDueTokensAccrued;
   18,  29:     uint public replenishmentPriceBps;
-  19     :-    mapping(address => uint256) public balances;
-  20     :-    mapping(address => mapping(address => uint256))
+       30:+    mapping(address => UserInfo) public userInfo;
   21,  31:     uint256 internal immutable INITIAL_CHAIN_ID;
   22,  32:     bytes32 internal immutable INITIAL_DOMAIN_SEPARA
-  23     :-    mapping(address => uint256) public nonces;
   24,  33:     mapping (address => bool) public minters;
   25,  34:     mapping (address => bool) public markets;
-  26     :-    mapping (address => uint) public debts; // user
-  27     :-    mapping (address => uint) public dueTokensAccrue
-  28     :-    mapping (address => uint) public lastUpdated; //
   29,  35:
   30,  36:     constructor(
   31,  37:         uint _replenishmentPriceBps,
@@ -118,10 +124,10 @@ contract DolaBorrowingRights {
  118, 124:     @return uint representing the balance of the use
  119, 125:     */
  120, 126:     function balanceOf(address user) public view ret
-  121     :-        uint debt = debts[user];
-  122     :-        uint accrued = (block.timestamp - lastUpdate
-  123     :-        if(dueTokensAccrued[user] + accrued > balanc
-  124     :-        return balances[user] - dueTokensAccrued[use
+      127:+        uint debt = userInfo[user].debts;
+      128:+        uint accrued = (block.timestamp - userInfo[u
+      129:+        if(userInfo[user].dueTokensAccrued + accruec
+      130:+        return userInfo[user].balances - userInfo[us
  125, 131:     }
  126, 132:
  127, 133:     /**
@@ -131,10 +137,10 @@ contract DolaBorrowingRights {
  131, 137:     @return uint representing the deficit of the use
  132, 138:     */
  133, 139:     function deficitOf(address user) public view ret
-  134     :-        uint debt = debts[user];
-  135     :-        uint accrued = (block.timestamp - lastUpdate
-  136     :-        if(dueTokensAccrued[user] + accrued < balanc
-  137     :-        return dueTokensAccrued[user] + accrued - ba
+      140:+        uint debt = userInfo[user].debts;
+      141:+        uint accrued = (block.timestamp - userInfo[u
+      142:+        if(userInfo[user].dueTokensAccrued + accruec
+      143:+        return userInfo[user].dueTokensAccrued + acc
```

```
  138, 144:     }
  139, 145:
  140, 146:     /**
@@ -144,9 +150,9 @@ contract DolaBorrowingRights {
  144, 150:     @return Returns a signed int of the user's balar
  145, 151:     */
  146, 152:     function signedBalanceOf(address user) public vi
- 147     :-        uint debt = debts[user];
- 148     :-        uint accrued = (block.timestamp - lastUpdate
- 149     :-        return int(balances[user]) - int(dueTokensAc
+      153:+        uint debt = userInfo[user].debts;
+      154:+        uint accrued = (block.timestamp - userInfo[u
+      155:+        return int(userInfo[user].balances) - int(us
  150, 156:     }
  151, 157:
  152, 158:     /**
@@ -169,9 +175,9 @@ contract DolaBorrowingRights {
  169, 175:     */
  170, 176:     function transfer(address to, uint256 amount) pu
  171, 177:         require(balanceOf(msg.sender) >= amount, "Ir
- 172     :-        balances[msg.sender] -= amount;
+      178:+        userInfo[msg.sender].balances -= amount;
  173, 179:         unchecked {
- 174     :-            balances[to] += amount;
+      180:+            userInfo[to].balances += amount;
  175, 181:         }
  176, 182:         emit Transfer(msg.sender, to, amount);
  177, 183:         return true;
@@ -193,9 +199,9 @@ contract DolaBorrowingRights {
  193, 199:         uint256 allowed = allowance[from][msg.sender
  194, 200:         if (allowed != type(uint256).max) allowance[
  195, 201:         require(balanceOf(from) >= amount, "Insuffic
- 196     :-        balances[from] -= amount;
+      202:+        userInfo[from].balances -= amount;
  197, 203:         unchecked {
- 198     :-            balances[to] += amount;
+      204:+            userInfo[to].balances += amount;
  199, 205:         }
  200, 206:         emit Transfer(from, to, amount);
  201, 207:         return true;
@@ -236,7 +242,7 @@ contract DolaBorrowingRights {
  236, 242:                                 owner,
  237, 243:                                 spender,
  238, 244:                                 value,
- 239     :-                                nonces[owner]++,
+      245:+                                userInfo[owner].nonc
```

```
  240, 246:                                 deadline
  241, 247:                             )
  242, 248:                         )
@@ -256,7 +262,7 @@ contract DolaBorrowingRights {
  256, 262:     @notice Function for invalidating the nonce of a
  257, 263:     */
  258, 264:     function invalidateNonce() public {
- 259     :-        nonces[msg.sender]++;
+      265:+        userInfo[msg.sender].nonce++;
  260, 266:     }
  261, 267:
  262, 268:     function DOMAIN_SEPARATOR() public view virtual
@@ -282,12 +288,12 @@ contract DolaBorrowingRights {
  282, 288:     @param user The address of the user to accrue DB
  283, 289:     */
  284, 290:     function accrueDueTokens(address user) public {
- 285     :-        uint debt = debts[user];
- 286     :-        if(lastUpdated[user] == block.timestamp) ret
- 287     :-        uint accrued = (block.timestamp - lastUpdate
- 288     :-        dueTokensAccrued[user] += accrued;
+      291:+        uint debt = userInfo[user].debts;
+      292:+        if(userInfo[user].lastUpdated == block.times
+      293:+        uint accrued = (block.timestamp - userInfo[u
+      294:+        userInfo[user].dueTokensAccrued += accrued;
  289, 295:         totalDueTokensAccrued += accrued;
- 290     :-        lastUpdated[user] = block.timestamp;
+      296:+        userInfo[user].lastUpdated = block.timestamp
  291, 297:         emit Transfer(user, address(0), accrued);
  292, 298:     }
  293, 299:
@@ -301,7 +307,7 @@ contract DolaBorrowingRights {
  301, 307:         require(markets[msg.sender], "Only markets c
  302, 308:         accrueDueTokens(user);
  303, 309:         require(deficitOf(user) == 0, "DBR Deficit")
- 304     :-        debts[user] += additionalDebt;
+      310:+        userInfo[user].debts += additionalDebt;
  305, 311:     }
  306, 312:
  307, 313:     /**
@@ -313,7 +319,7 @@ contract DolaBorrowingRights {
  313, 319:     function onRepay(address user, uint repaidDebt)
  314, 320:         require(markets[msg.sender], "Only markets c
  315, 321:         accrueDueTokens(user);
- 316     :-        debts[user] -= repaidDebt;
+      322:+        userInfo[user].debts -= repaidDebt;
  317, 323:     }
```

```
  318, 324:
  319, 325:     /**
@@ -329,7 +335,7 @@ contract DolaBorrowingRights {
  329, 335:         require(deficit >= amount, "Amount > deficit
  330, 336:         uint replenishmentCost = amount * replenishm
  331, 337:         accrueDueTokens(user);
- 332     :-        debts[user] += replenishmentCost;
+      338:+        userInfo[user].debts += replenishmentCost;
  333, 339:         _mint(user, amount);
  334, 340:     }
  335, 341:
@@ -359,7 +365,7 @@ contract DolaBorrowingRights {
  359, 365:     function _mint(address to, uint256 amount) inter
  360, 366:         _totalSupply += amount;
  361, 367:         unchecked {
- 362     :-            balances[to] += amount;
+      368:+            userInfo[to].balances += amount;
  363, 369:         }
  364, 370:         emit Transfer(address(0), to, amount);
  365, 371:     }
@@ -371,7 +377,7 @@ contract DolaBorrowingRights {
  371, 377:     */
  372, 378:     function _burn(address from, uint256 amount) int
  373, 379:         require(balanceOf(from) >= amount, "Insuffic
- 374     :-        balances[from] -= amount;
+      380:+        userInfo[from].balances -= amount;
  375, 381:         unchecked {
  376, 382:             _totalSupply -= amount;
  377, 383:         }
diff --git a/src/test/DBR.t.sol b/src/test/DBR.t.sol
index 3988cf7..754bf7f 100644
--- a/src/test/DBR.t.sol
+++ b/src/test/DBR.t.sol
@@ -145,17 +145,19 @@ contract DBRTest is FiRMTest {
  145, 145:     }
  146, 146:
  147, 147:     function test_invalidateNonce() public {
- 148     :-        assertEq(dbr.nonces(user), 0, "User nonce sh
+      148:+        (, uint256 nonce,,,) = dbr.userInfo(user);
+      149:+        assertEq(nonce, 0, "User nonce should be uni
  149, 150:
  150, 151:         vm.startPrank(user);
  151, 152:         dbr.invalidateNonce();
  152, 153:
- 153     :-        assertEq(dbr.nonces(user), 1, "User nonce wa
+      154:+        (,nonce,,,) = dbr.userInfo(user);
```

```
+       155:+         assertEq(nonce, 1, "User nonce was not inval
  154,  156:      }
  155,  157:
  156,  158:      function test_approve_increasesAllowanceByAmount
  157,  159:          uint amount = 100e18;
- 158      :-
+       160:+
  159,  161:          assertEq(dbr.allowance(user, gov), 0, "Allow
  160,  162:
  161,  163:          vm.startPrank(user);
```

[05] Expression can be unchecked when overflow is not possible (6 instances)

Deployment. Gas Saved: **20 220**

Minimum Method Call. Gas Saved: **410**

Average Method Call. Gas Saved: **4 630**

Maximum Method Call. Gas Saved: **1 354**

Overall gas change: **-6 233 (-5.326%)**

src/DBR.sol:110, 124, 137, 259

```
diff --git a/src/DBR.sol b/src/DBR.sol
index aab6daf..0781c97 100644
--- a/src/DBR.sol
+++ b/src/DBR.sol
@@ -107,8 +107,10 @@ contract DolaBorrowingRights {
  107,  107:      @return uint representing the total supply of DB
  108,  108:      */
  109,  109:      function totalSupply() public view returns (uint
- 110      :-         if(totalDueTokensAccrued > _totalSupply) ret
- 111      :-         return _totalSupply - totalDueTokensAccrued;
+       110:+         unchecked {
+       111:+             if(totalDueTokensAccrued > _totalSupply)
+       112:+             return _totalSupply - totalDueTokensAccr
+       113:+         }
  112,  114:      }
```

```
  113, 115:
  114, 116:     /**
@@ -121,7 +123,7 @@ contract DolaBorrowingRights {
  121, 123:         uint debt = debts[user];
  122, 124:         uint accrued = (block.timestamp - lastUpdate
  123, 125:         if(dueTokensAccrued[user] + accrued > balanc
- 124     :-        return balances[user] - dueTokensAccrued[use
+      126:+        unchecked { return balances[user] - dueToker
  125, 127:     }
  126, 128:
  127, 129:     /**
@@ -134,7 +136,7 @@ contract DolaBorrowingRights {
  134, 136:         uint debt = debts[user];
  135, 137:         uint accrued = (block.timestamp - lastUpdate
  136, 138:         if(dueTokensAccrued[user] + accrued < balanc
- 137     :-        return dueTokensAccrued[user] + accrued - ba
+      139:+        unchecked { return dueTokensAccrued[user] +
  138, 140:     }
  139, 141:
  140, 142:     /**
@@ -256,7 +258,7 @@ contract DolaBorrowingRights {
  256, 258:     @notice Function for invalidating the nonce of a
  257, 259:     */
  258, 260:     function invalidateNonce() public {
- 259     :-        nonces[msg.sender]++;
+      261:+        unchecked { nonces[msg.sender]++; }
  260, 262:     }
  261, 263:
  262, 264:     function DOMAIN_SEPARATOR() public view virtual
```

src/Fed.sol:124

```
diff --git a/src/Fed.sol b/src/Fed.sol
index 1e819bb..b57b444 100644
--- a/src/Fed.sol
+++ b/src/Fed.sol
@@ -121,7 +121,7 @@ contract Fed {
  121, 121:         uint marketValue = dola.balanceOf(address(ma
  122, 122:         uint supply = supplies[market];
  123, 123:         if(supply >= marketValue) return 0;
- 124     :-        return marketValue - supply;
+      124:+        unchecked { return marketValue - supply; }
  125, 125:     }
```

```
  126, 126:
  127, 127:     /**
```

src/Market.sol:521

```
diff --git a/src/Market.sol b/src/Market.sol
index 9585b85..293bbb6 100644
--- a/src/Market.sol
+++ b/src/Market.sol
@@ -518,7 +518,7 @@ contract Market {
  518, 518:     @notice Function for incrementing the nonce of t
  519, 519:     */
  520, 520:     function invalidateNonce() public {
- 521     :-         nonces[msg.sender]++;
+      521:+         unchecked { nonces[msg.sender]++; }
  522, 522:     }
  523, 523:
  524, 524:     /**
```

[06] State variables can be packed into fewer storage slots (1 instance)

Deployment. Gas Saved: **-5 008**

Minimum Method Call. Gas Saved: **1 911**

Average Method Call. Gas Saved: **15 525**

Maximum Method Call. Gas Saved: **20 972**

Overall gas change: **-62 419 (-69.524%)**

If variables occupying the same slot are both written the same function or by the constructor, avoids a separate Gsset (20000 gas). Reads of the variables can also be cheaper

uint256(32), mapping(32), address(20), bool(1)

src/Market.sol:53

```
diff --git a/src/Market.sol b/src/Market.sol
index 9585b85..6141e5c 100644
--- a/src/Market.sol
+++ b/src/Market.sol
@@ -36,6 +36,7 @@ interface IBorrowController {
   36,  36: contract Market {
   37,  37:
   38,  38:     address public gov;
+       39:+    bool public borrowPaused;
   39,  40:     address public lender;
   40,  41:     address public pauseGuardian;
   41,  42:     address public immutable escrowImplementation;
@@ -50,7 +51,6 @@ contract Market {
   50,  51:     uint public liquidationFeeBps;
   51,  52:     uint public liquidationFactorBps = 5000; // 50%
   52,  53:     bool immutable callOnDepositCallback;
-  53     :-    bool public borrowPaused;
   54,  54:     uint public totalDebt;
   55,  55:     uint256 internal immutable INITIAL_CHAIN_ID;
   56,  56:     bytes32 internal immutable INITIAL_DOMAIN_SEPARA
```

[07] Refactoring similar statements (1 instance)

Deployment. Gas Saved: **18 422**

Minimum Method Call. Gas Saved: **-18**

Average Method Call. Gas Saved: **-11**

Maximum Method Call. Gas Saved: **6**

Overall gas change: **4 876 (7.739%)**

src/Market.sol:213

```
diff --git a/src/Market.sol b/src/Market.sol
index 9585b85..da295e5 100644
--- a/src/Market.sol
```

```
+++ b/src/Market.sol
@@ -210,11 +210,9 @@ contract Market {
  210, 210:     @param _value Boolean representing the state pau
  211, 211:     */
  212, 212:     function pauseBorrows(bool _value) public {
- 213     :-        if(_value) {
- 214     :-            require(msg.sender == pauseGuardian || m
- 215     :-        } else {
- 216     :-            require(msg.sender == gov, "Only governa
- 217     :-        }
+      213:+        require(
+      214:+            ( _value && msg.sender == pauseGuardian)
+      215:+            "Only pause guardian or governance can p
  218, 216:         borrowPaused = _value;
  219, 217:     }
  220, 218:
diff --git a/src/test/Market.t.sol b/src/test/Market.t.sol
index 8992ab9..86af449 100644
--- a/src/test/Market.t.sol
+++ b/src/test/Market.t.sol
@@ -16,7 +16,7 @@ import "./mocks/BorrowContract.sol";
   16,  16: import {EthFeed} from "./mocks/EthFeed.sol";
   17,  17:
   18,  18: contract MarketTest is FiRMTest {
-  19     :-    bytes onlyGovUnpause = "Only governance can unpa
+      19:+    bytes onlyGovUnpause = "Only pause guardian or g
   20,  20:     bytes onlyPauseGuardianOrGov = "Only pause guarc
   21,  21:
   22,  22:     BorrowContract borrowContract;
```

[08] Better algorithm for underflow check (3 instances)

Deployment. Gas Saved: **12 613**

Minimum Method Call. Gas Saved: **656**

Average Method Call. Gas Saved: **8 332**

Maximum Method Call. Gas Saved: **3 741**

Overall gas change: **-18 048 (-15.981%)**

```
diff --git a/src/DBR.sol b/src/DBR.sol
index aab6daf..bff9fef 100644
--- a/src/DBR.sol
+++ b/src/DBR.sol
@@ -104,37 +104,39 @@ contract DolaBorrowingRights {
  104, 104:     /**
  105, 105:     @notice Get the total supply of DBR tokens.
  106, 106:     @dev The total supply is calculated as the diffe
- 107     :-    @return uint representing the total supply of DB
+      107:+    @return ret uint representing the total supply o
  108, 108:     */
- 109     :-    function totalSupply() public view returns (uint
- 110     :-        if(totalDueTokensAccrued > _totalSupply) ret
- 111     :-        return _totalSupply - totalDueTokensAccrued;
+      109:+    function totalSupply() public view returns (uint
+      110:+        unchecked { ret = _totalSupply - totalDueTok
+      111:+        if(ret > _totalSupply) return 0;
  112, 112:     }
  113, 113:
  114, 114:     /**
  115, 115:     @notice Get the DBR balance of an address. Will
  116, 116:     @dev The balance of a user is calculated as the
  117, 117:     @param user Address of the user.
- 118     :-    @return uint representing the balance of the use
+      118:+    @return ret uint representing the balance of the
  119, 119:     */
- 120     :-    function balanceOf(address user) public view ret
+      120:+    function balanceOf(address user) public view ret
  121, 121:         uint debt = debts[user];
  122, 122:         uint accrued = (block.timestamp - lastUpdate
- 123     :-        if(dueTokensAccrued[user] + accrued > balanc
- 124     :-        return balances[user] - dueTokensAccrued[use
+      123:+        uint mid = dueTokensAccrued[user] + accrued;
+      124:+        unchecked { ret = balances[user] - mid; }
+      125:+        if(ret > balances[user]) return 0;
  125, 126:     }
  126, 127:
  127, 128:     /**
  128, 129:     @notice Get the DBR deficit of an address. Will
  129, 130:     @dev The deficit of a user is calculated as the
  130, 131:     @param user Address of the user.
- 131     :-    @return uint representing the deficit of the use
+      132:+    @return ret uint representing the deficit of the
```

```
  132, 133:     */
- 133     :-    function deficitOf(address user) public view ret
+      134:+    function deficitOf(address user) public view ret
  134, 135:         uint debt = debts[user];
  135, 136:         uint accrued = (block.timestamp - lastUpdate
- 136     :-        if(dueTokensAccrued[user] + accrued < balanc
- 137     :-        return dueTokensAccrued[user] + accrued - ba
+      137:+        uint mid = dueTokensAccrued[user] + accrued;
+      138:+        unchecked { ret = mid - balances[user]; }
+      139:+        if(mid < ret) return 0;
  138, 140:     }
  139, 141:
  140, 142:     /**
```

[09] `x = x + y` is cheaper than `x += y` (12 instances)

Deployment. Gas Saved: **11 214**

Minimum Method Call. Gas Saved: **180**

Average Method Call. Gas Saved: **468**

Maximum Method Call. Gas Saved: **616**

Overall gas change: **-5 325 (-1.318%)**

src/DBR.sol:174, 196, 289, 360, 362, 376

```
diff --git a/src/DBR.sol b/src/DBR.sol
index aab6daf..c02b782 100644
--- a/src/DBR.sol
+++ b/src/DBR.sol
@@ -171,7 +171,7 @@ contract DolaBorrowingRights {
  171, 171:         require(balanceOf(msg.sender) >= amount, "Ir
  172, 172:         balances[msg.sender] -= amount;
  173, 173:         unchecked {
- 174     :-            balances[to] += amount;
+      174:+            balances[to] = balances[to] + amount;
  175, 175:         }
  176, 176:         emit Transfer(msg.sender, to, amount);
  177, 177:         return true;
```

```
@@ -193,7 +193,7 @@ contract DolaBorrowingRights {
  193, 193:         uint256 allowed = allowance[from][msg.sender
  194, 194:         if (allowed != type(uint256).max) allowance[
  195, 195:         require(balanceOf(from) >= amount, "Insuffic
- 196     :-         balances[from] -= amount;
+      196:+         balances[from] = balances[from] - amount;
  197, 197:         unchecked {
  198, 198:             balances[to] += amount;
  199, 199:         }
@@ -286,7 +286,7 @@ contract DolaBorrowingRights {
  286, 286:         if(lastUpdated[user] == block.timestamp) ret
  287, 287:         uint accrued = (block.timestamp - lastUpdate
  288, 288:         dueTokensAccrued[user] += accrued;
- 289     :-         totalDueTokensAccrued += accrued;
+      289:+         totalDueTokensAccrued = totalDueTokensAccrue
  290, 290:         lastUpdated[user] = block.timestamp;
  291, 291:         emit Transfer(user, address(0), accrued);
  292, 292:     }
@@ -357,9 +357,9 @@ contract DolaBorrowingRights {
  357, 357:     @param amount Amount of DBR to mint.
  358, 358:     */
  359, 359:     function _mint(address to, uint256 amount) inter
- 360     :-         _totalSupply += amount;
+      360:+         _totalSupply = _totalSupply + amount;
  361, 361:         unchecked {
- 362     :-             balances[to] += amount;
+      362:+             balances[to] = balances[to] + amount;
  363, 363:         }
  364, 364:         emit Transfer(address(0), to, amount);
  365, 365:     }
@@ -373,7 +373,7 @@ contract DolaBorrowingRights {
  373, 373:         require(balanceOf(from) >= amount, "Insuffic
  374, 374:         balances[from] -= amount;
  375, 375:         unchecked {
- 376     :-             _totalSupply -= amount;
+      376:+             _totalSupply = _totalSupply - amount;
  377, 377:         }
  378, 378:         emit Transfer(from, address(0), amount);
  379, 379:     }
```

src/Market.sol:395, 397, 535, 568, 598, 600

```
diff --git a/src/Market.sol b/src/Market.sol
```

```
index 9585b85..bc0ff93 100644
--- a/src/Market.sol
+++ b/src/Market.sol
@@ -392,9 +392,9 @@ contract Market {
  392, 392:             require(borrowController.borrowAllowed(m
  393, 393:         }
  394, 394:         uint credit = getCreditLimitInternal(borrowe
- 395     :-        debts[borrower] += amount;
+      395:+       debts[borrower] = debts[borrower] + amount;
  396, 396:         require(credit >= debts[borrower], "Exceedec
- 397     :-        totalDebt += amount;
+      397:+       totalDebt = totalDebt + amount;
  398, 398:         dbr.onBorrow(borrower, amount);
  399, 399:         dola.transfer(to, amount);
  400, 400:         emit Borrow(borrower, amount);
@@ -532,7 +532,7 @@ contract Market {
  532, 532:         uint debt = debts[user];
  533, 533:         require(debt >= amount, "Insufficient debt")
  534, 534:         debts[user] -= amount;
- 535     :-        totalDebt -= amount;
+      535:+       totalDebt = totalDebt - amount;
  536, 536:         dbr.onRepay(user, amount);
  537, 537:         dola.transferFrom(msg.sender, address(this),
  538, 538:         emit Repay(user, msg.sender, amount);
@@ -565,7 +565,7 @@ contract Market {
  565, 565:         debts[user] += replenishmentCost;
  566, 566:         uint collateralValue = getCollateralValueInt
  567, 567:         require(collateralValue >= debts[user], "Exc
- 568     :-        totalDebt += replenishmentCost;
+      568:+       totalDebt = totalDebt + replenishmentCost;
  569, 569:         dbr.onForceReplenish(user, amount);
  570, 570:         dola.transfer(msg.sender, replenisherReward)
  571, 571:         emit ForceReplenish(user, msg.sender, amount
@@ -595,9 +595,9 @@ contract Market {
  595, 595:         require(repaidDebt <= debt * liquidationFact
  596, 596:         uint price = oracle.getPrice(address(collate
  597, 597:         uint liquidatorReward = repaidDebt * 1 ether
- 598     :-        liquidatorReward += liquidatorReward * liqui
+      598:+       liquidatorReward = liquidatorReward + liquic
  599, 599:         debts[user] -= repaidDebt;
- 600     :-        totalDebt -= repaidDebt;
+      600:+       totalDebt = totalDebt - repaidDebt;
  601, 601:         dbr.onRepay(user, repaidDebt);
  602, 602:         dola.transferFrom(msg.sender, address(this),
  603, 603:         IEscrow escrow = predictEscrow(user);
```

[10] `internal` functions only called once can be inlined to save gas (1 instance)

Deployment. Gas Saved: **5 207**

Minimum Method Call. Gas Saved: **67**

Average Method Call. Gas Saved: **47**

Maximum Method Call. Gas Saved: **24**

Overall gas change: -137 (-0.154%)

src/DBR.sol:341

```
diff --git a/src/DBR.sol b/src/DBR.sol
index aab6daf..a357f92 100644
--- a/src/DBR.sol
+++ b/src/DBR.sol
@@ -338,7 +338,12 @@ contract DolaBorrowingRights {
  338, 338:     @param amount Amount to be burned
  339, 339:     */
  340, 340:     function burn(uint amount) public {
- 341     :-        _burn(msg.sender, amount);
+      341:+        require(balanceOf(msg.sender) >= amount, "Ir
+      342:+        balances[msg.sender] -= amount;
+      343:+        unchecked {
+      344:+            _totalSupply -= amount;
+      345:+        }
+      346:+        emit Transfer(msg.sender, address(0), amount
  342, 347:     }
  343, 348:
  344, 349:     /**
@@ -364,20 +369,6 @@ contract DolaBorrowingRights {
  364, 369:         emit Transfer(address(0), to, amount);
  365, 370:     }
  366, 371:
- 367     :-    /**
- 368     :-    @notice Internal function for burning DBR.
- 369     :-    @param from Address to burn DBR from.
- 370     :-    @param amount Amount of DBR to be burned.
```

```
- 371     :-    */
- 372     :-    function _burn(address from, uint256 amount) int
- 373     :-        require(balanceOf(from) >= amount, "Insuffic
- 374     :-        balances[from] -= amount;
- 375     :-        unchecked {
- 376     :-            _totalSupply -= amount;
- 377     :-        }
- 378     :-        emit Transfer(from, address(0), amount);
- 379     :-    }
- 380     :-
  381, 372:     event Transfer(address indexed from, address inc
  382, 373:     event Approval(address indexed owner, address ir
  383, 374:     event AddMinter(address indexed minter);
```

[11] State variables should be cached in stack variables rather than re-reading them from storage (2 instances)

Deployment. Gas Saved: **5 007**

Minimum Method Call. Gas Saved: **478**

Average Method Call. Gas Saved: **1 117**

Maximum Method Call. Gas Saved: **1 423**

Overall gas change: **-6 231 (-1.618%)**

src/DBR.sol:286

```
diff --git a/src/DBR.sol b/src/DBR.sol
index aab6daf..c70fcd7 100644
--- a/src/DBR.sol
+++ b/src/DBR.sol
@@ -283,8 +283,9 @@ contract DolaBorrowingRights {
  283, 283:     */
  284, 284:     function accrueDueTokens(address user) public {
  285, 285:         uint debt = debts[user];
- 286     :-        if(lastUpdated[user] == block.timestamp) ret
- 287     :-        uint accrued = (block.timestamp - lastUpdate
+      286:+        uint _lastUpdated = lastUpdated[user];
+      287:+        if(_lastUpdated == block.timestamp) return;
```

```
+      288:+         uint accrued = (block.timestamp - _lastUpdat
  288, 289:          dueTokensAccrued[user] += accrued;
  289, 290:          totalDueTokensAccrued += accrued;
  290, 291:          lastUpdated[user] = block.timestamp;
```

src/Market.sol:391

```
diff --git a/src/Market.sol b/src/Market.sol
index 9585b85..5f3264d 100644
--- a/src/Market.sol
+++ b/src/Market.sol
@@ -388,8 +388,9 @@ contract Market {
  388, 388:      */
  389, 389:     function borrowInternal(address borrower, addres
  390, 390:         require(!borrowPaused, "Borrowing is paused'
- 391     :-         if(borrowController != IBorrowController(adc
- 392     :-             require(borrowController.borrowAllowed(m
+      391:+         IBorrowController _borrowController = borrow
+      392:+         if(_borrowController != IBorrowController(ac
+      393:+             require(_borrowController.borrowAllowed(
  393, 394:          }
  394, 395:          uint credit = getCreditLimitInternal(borrowe
  395, 396:          debts[borrower] += amount;
```

Overall gas savings

Deployment. Gas Saved: **416 802**

Minimum Method Call. Gas Saved: **3 423**

Average Method Call. Gas Saved: **15 773**

Maximum Method Call. Gas Saved: **18 283**

Overall gas change: **-84 866 (-67.204%)**

Please see warden's [original submission](#) for full details and diff.

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