Venus Prime
Findings & Analysis Report

2023-11-29

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 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 outlined in this document, C4 conducted an analysis of the Venus Prime smart contract system written in Solidity. The audit took place between September 28—October 4 2023.

Wardens

130 Wardens contributed reports to Venus Prime:

1. Testerbot
2. 0xTheC0der
3. 0xDetermination
4. Brenzee
5. ether_sky
6. SpicyMeatball
7. Breeje
8. tapir

9. santipu_
10. sces60107
11. ast3ros
12. pep7siup
13. ThreeSigma (0x73696d616f, 0xCarolina, EduCatarino, and SolidityDev99)
14. said
15. 3agle
16. rokinot
17. PwnStars (qbs and sakshamguruji)
18. neumo
19. blutorque
20. Pessimistic (oleggatttor, yhtyyar, and PavelCore)
21. DavidGiladi
22. oakcobalt
23. hals
24. bin2chen
25. DeFiHackLabs (AkshaySrivastav, Cache_and_Burn, IceBear, Ronin, Sm4rty, SunSec, sashik_eth, zuhaibmohd, and ret2basic)
26. Norah
27. seerether
28. turvy_fuzz
29. dirk_y
30. deadrxsezzz
31. 0xprinc
32. 0x3b
33. deth
34. J4X
35. Satyam_Sharma
36. gkrastenov

37. merlin
38. Flora
39. KrisApostolov
40. berlin-101
41. twicek
42. 0xpiken
43. rvierdiiev
44. HChang26
45. mahdirostami
46. lsaudit
47. aycozynfada
48. sl1
49. 0xhacksmithh
50. pavankv
51. Bauchibred
52. maanas
53. josephdara
54. 0xweb3boy
55. al88nsk
56. xAriextz
57. pina
58. SBSecurity (Slavcheww and Blckhv)
59. dethera
60. 0xblackskull
61. tsvetanovv
62. ADM
63. debo
64. ArmedGoose
65. jkoppel

66. OxWaitress
67. radev_sw
68. hunter_w3b
69. kaveyjoe
70. versiyonbir
71. jamshed
72. pontifex
73. hihen
74. OxScourgedev
75. Maroutis
76. inzinko
77. ge6a
78. OxMosh
79. btk
80. Oxdice91
81. Fulum
82. imare
83. Aymen0909
84. SPYBOY
85. jnforja
86. alexweb3
87. Mirror
88. OxTiwa
89. d3e4
90. tonisives
91. MohammedRizwan
92. Tricko
93. Oxfusion
94. glcanvas

95. lotux
96. y4y
97. sashik_eth
98. Krace
99. Daniel526
100. squeaky_cactus
101. joaovwfreire
102. peanuts
103. vagrant
104. nobody2018
105. nisedo
106. IceBear
107. terrancrypt
108. Hama
109. n1punp
110. nadin
111. e0d1n
112. kutugu
113. TangYuanShen
114. orion
115. ptsanev

This audit was judged by 0xDjango.

Final report assembled by PaperParachute.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 3 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 88 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 11 reports recommending gas optimizations.

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

Scope

The code under review can be found within the [C4 Venus Prime repository](), and is composed of 7 smart contracts written in the Solidity programming language and includes 1039 lines of Solidity code.

In addition to the known issues identified by the project team, a Code4rena bot race was conducted at the start of the audit. The winning bot, **Tera Bot** from warden [kn0t](), generated the [Automated Findings report]() and all findings therein were classified as out of scope.

Severity Criteria

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

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

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

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

High Risk Findings (3)

[H-01] Prime.sol - User can claim Prime token without having

any staked XVS, because his `stakedAt` isn’t reset whenever he is issued an irrevocable token

Submitted by deth, *also found by* rokinot, gkrastenov, merlin, Testerbot, 0xDetermination, 3agle, aycozynfada, Flora, KrisApostolov, berlin-101, santipu_, twicek, sl1, 0xpiken, Brenzee, rvierdiiev, tapir, HChang26, Satyam_Sharma, mahdirostami, *and* said

Whenever a new Prime token is created, the users `stakedAt` is reset to 0. This happens when the user `claim` a revocable token and when he is `issue` a revocable token, but it does not happen when a user is `issue` an irrevocable token.

This is `issue()`

```
    function issue(bool isIrrevocable, address[] calldata users) ext
        _checkAccessAllowed("issue(bool,address[])");

        if (isIrrevocable) {
            for (uint256 i = 0; i < users.length; ) {
                Token storage userToken = tokens[users[i]];
                if (userToken.exists && !userToken.isIrrevocable
                    _upgrade(users[i]);
                } else {
                    // We don't reset here.
                    _mint(true, users[i]);
                    _initializeMarkets(users[i]);
                }

                unchecked {
                    i++;
                }
            }
        } else {
            for (uint256 i = 0; i < users.length; ) {
                _mint(false, users[i]);
                _initializeMarkets(users[i]);

                // We reset stakedAt here
                delete stakedAt[users[i]];

                unchecked {
                    i++;
                    }
                }
            }
        }
```

We can see that when a revocable token is issued and minted the user's `stakedAt` is reset to 0. Whenever a user's token is upgraded, his `stakedAt` has already been reset to 0 inside `claim`.

```
    function claim() external {
            if (stakedAt[msg.sender] == 0) revert IneligibleToClaim
            if (block.timestamp - stakedAt[msg.sender] < STAKING_PEF

            // We reset stakedAt here
            stakedAt[msg.sender] = 0;

            _mint(false, msg.sender);
            _initializeMarkets(msg.sender);
        }
```

The only one time when we don't reset the user's `stakedAt` and it's when he is issued an irrevocable token.

Let's see an example and see why this is a problem:

1. Alice deposits 10k XVS.
2. The protocol/DAO/admin decides to issue Alice an irrevocable prime token, because she deposited such a large amount of tokens. Keep in mind that the 90 day staking period still hasn't passed and her `stakedAt` is the original time that she deposited 10k XVS.
3. Time passes and Alice decides to withdraw her entire XVS, so now she has 0 XVS. Her token isn't burned as she has an irrevocable token.
4. Even more time passes and the protocol/DAO/admin decides to burn Alice's irrevocable token because she is inactive.
5. EVEN more time passes and Alice returns to the protocol and instead of depositing anything, she calls `claim`.

Her tx goes through, since her `stakedAt` wasn't reset to 0 when she got issued her irrevocable token.

This way, Alice claimed a revocable token without having any XVS staked in the contract.

Proof of Concept

Add the following line at the top of `tests/hardhat/Prime/Prime.ts`. We'll use this to simulate time passing

```
import { time } from "@nomicfoundation/hardhat-network-helpers";
```

Paste the following inside `tests/hardhat/Prime/Prime.ts` and run `npx hardhat test tests/hardhat/Prime/Prime.ts`.

```
it.only("User can get Prime token without any XVS staked", async
      // User1 deposits 10k XVS
      await xvs.transfer(await user1.getAddress(), parseUnits("1
      await xvs.connect(user1).approve(xvsVault.address, parseUr
      await xvsVault.connect(user1).deposit(xvs.address, 0, pars
      let userInfo = await xvsVault.getUserInfo(xvs.address, 0,
      expect(userInfo.amount).to.eq(parseUnits("10000", 18));

      // Venus decides to issue an irrevocable Prime token to Us
      // Note that the 90 day staking period still hasn't passec
      await prime.issue(true, [user1.getAddress()]);
      let token = await prime.tokens(user1.getAddress());
      expect(token.exists).to.be.equal(true);
      expect(token.isIrrevocable).to.be.equal(true);

      // User1 withdraws her entire balance XVS
      await xvsVault.connect(user1).requestWithdrawal(xvs.addres
      userInfo = await xvsVault.getUserInfo(xvs.address, 0, user
      expect(userInfo.pendingWithdrawals).to.eq(parseUnits("100(

      // User1's Prime token gets burned by protocol
      await prime.burn(user1.getAddress());
      token = await prime.tokens(user1.getAddress());
      expect(token.exists).to.be.equal(false);
      expect(token.isIrrevocable).to.be.equal(false);
```

```
        // 100 days pass
        await time.increase(8640000);

        // User1 can claim a revocable Prime token without any XVS
        expect(prime.stakedAt(await user1.getAddress())).to.not.be

        await prime.connect(user1).claim();
        token = await prime.tokens(user1.getAddress());
        expect(token.exists).to.be.equal(true);
        expect(token.isIrrevocable).to.be.equal(false);
      });
```

If you are having trouble running the test, this change might fix it. Inside `Prime.sol`, `burn()` remove the access control from the function. This doesn't change the attack and the test outcome.

```
    function burn(address user) external {
            // _checkAccessAllowed("burn(address)");
            _burn(user);
        }
```

Tools Used

Hardhat

Recommended Mitigation Steps

Reset the user's `stakedAt` whenever he is issued an irrevocable token.

```
    function issue(bool isIrrevocable, address[] calldata users) ext
            _checkAccessAllowed("issue(bool,address[])");

            if (isIrrevocable) {
                for (uint256 i = 0; i < users.length; ) {
                    Token storage userToken = tokens[users[i]];
                    if (userToken.exists && !userToken.isIrrevocable
                        _upgrade(users[i]);
                    } else {
                        _mint(true, users[i]);
                        _initializeMarkets(users[i]);
                        delete stakedAt[users[i]];
                    }
              ...
```

chechu (Venus) confirmed and commented:

> Fixed here.

0xDjango (Judge) commented:

> Valid issue with serious impact to the protocol.

[H-02] A malicious user can avoid unfavorable score updates after alpha/multiplier changes, resulting in accrual of outsized rewards for the attacker at the expense of other users

Submitted by 0xDetermination, *also found by* hals, Testerbot, bin2chen, Pessimistic, rokinot, *ThreeSigma (*1, 2*)*, ether_sky, PwnStars, neumo, DeFiHackLabs, turvy_fuzz, Norah, dirk_y, deadrxsezzz, blutorque, SpicyMeatball, seerether, *and* said

https://github.com/code-423n4/2023-09-venus/blob/main/contracts/Tokens/Prime/Prime.sol#L397-L405

https://github.com/code-423n4/2023-09-venus/blob/main/contracts/Tokens/Prime/Prime.sol#L704-L756

https://github.com/code-423n4/2023-09-venus/blob/main/contracts/Tokens/Prime/Prime.sol#L623-L639

https://github.com/code-423n4/2023-09-venus/blob/main/contracts/Tokens/Prime/Prime.sol#L827-L833

https://github.com/code-423n4/2023-09-venus/blob/main/contracts/Tokens/Prime/Prime.sol#L200-L219 https://github.com/code-423n4/2023-09-venus/blob/main/tests/hardhat/Prime/Prime.ts#L294-L301

Please note: All functions/properties referred to are in the `Prime.sol` contract.

Impact

A malicious user can accrue outsized rewards at the expense of other users after `updateAlpha()` or `updateMultipliers()` is called.

Proof of Concept

An attacker can prevent their score from being updated and decreased after the protocol's alpha or multipliers change. This is done by manipulatively decreasing the value of `pendingScoreUpdates` , then ensuring that only other user scores are updated until `pendingScoreUpdates` reaches zero, at which point calls to `updateScores()` will revert with the error `NoScoreUpdatesRequired()` . This can be done via the attacker calling `updateScores()` to update other users' scores first and/or DoSing calls to `updateScores()` that would update the attacker's score (see the issue titled "DoS and gas griefing of Prime.updateScores()").

The core of this vulnerability is the attacker's ability to manipulate `pendingScoreUpdates` . Notice below that `claim()` , which is called to mint a user's Prime token, doesn't change the value of `pendingScoreUpdates` :

▶ Details

However, burning a token decrements `pendingScoreUpdates` . (Burning a token is done by withdrawing XVS from `XVSVault.sol` so that the resulting amount staked is below the minimum amount required to possess a Prime token.) Notice below:

```
    function _burn(address user) internal {
        ...
        _updateRoundAfterTokenBurned(user);

        emit Burn(user);
    }
    function _updateRoundAfterTokenBurned(address user) internal
        if (totalScoreUpdatesRequired > 0) totalScoreUpdatesRequ
        if (pendingScoreUpdates > 0 && !isScoreUpdated[nextScore
            pendingScoreUpdates--;
        }
    }
```

To inappropriately decrement the value of `pendingScoreUpdates` , the attacker can backrun the transaction updating the alpha/multiplier, minting and burning a Prime token (this requires the attacker to have staked the minimum amount of XVS 90 days in advance). If the number of Prime tokens minted is often at the max number of Prime tokens minted, the attacker could burn an existing token and then mint and burn a new one. Since the value of `!isScoreUpdated[nextScoreUpdateRoundId][user]` is default false, pendingScoreUpdates will be inappropriately decremented if the burned token was minted after the call to `updateMultipliers()` / `updateAlpha()` .

As aforementioned, the attacker can ensure that only other users' scores are updated until `pendingScoreUpdates` reaches zero, at which point further calls to `updateScores` will revert with the custom error `NoScoreUpdatesRequired()` .

Relevant code from `updateScores()` for reference:

```
    function updateScores(address[] memory users) external {
        if (pendingScoreUpdates == 0) revert NoScoreUpdatesRequi
        if (nextScoreUpdateRoundId == 0) revert NoScoreUpdatesRe

        for (uint256 i = 0; i < users.length; ) {
            ...
            pendingScoreUpdates--;
            isScoreUpdated[nextScoreUpdateRoundId][user] = true;

            unchecked {
                i++;
            }

            emit UserScoreUpdated(user);
        }
    }
```

As seen, the attacker's score can avoid being updated. This is signficant if a change in multiplier or alpha would decrease the attacker's score. Because rewards are distributed according to the user's score divided by the total score, the attacker can 'freeze' their score at a higher than appropriate value and accrue increased rewards at the cost of the other users in the market.

The attacker can also prevent score updates for other users. The attacker can 'freeze' a user's score that would otherwise increase after the alpha/multiplier changes, resulting in even greater rewards accrued for the attacker and denied from other users. This is because it is possible to decrease the value of `pendingScoreUpdates` by more than one if the attacker mints and burns more than one token after the alpha/multiplier is updated.

Math to support that a larger score results in greater reward accrual

Let a represent the attacker's score if it is properly updated after a change in alpha/multiplier, b represent the properly updated total score, and c represent the difference between the attacker's larger unupdated score and the attacker's smaller updated score. Clearly a, b, and c are positive with $a < b$. Consider the following inequality, which holds true since $a \< b$:

$\`frac{a+c}{b+c} > \frac{a}{b} \iff a+c > \frac{a(b+c)}{b} \iff a+c > a+\frac{ac}{b}$

Test

Paste and run the below test in the 'mint and burn' scenario in Prime.ts (line 302)

▶ Details

Tools Used

Hardhat

Recommended Mitigation Steps

Check if `pendingScoreUpdates` is nonzero when a token is minted, and increment it if so. This removes the attacker's ability to manipulate `pendingScoreUpdates` .

```
    function _mint(bool isIrrevocable, address user) internal {
        if (tokens[user].exists) revert IneligibleToClaim();
```

```
            tokens[user].exists = true;
            tokens[user].isIrrevocable = isIrrevocable;

            if (isIrrevocable) { //@Gas
                totalIrrevocable++;
            } else {
                totalRevocable++;
            }

            if (totalIrrevocable > irrevocableLimit || totalRevocabl
+           if (pendingScoreUpdates != 0) {unchecked{++pendingScoreU

            emit Mint(user, isIrrevocable);
        }
```

Further Considerations

The call to `updateMultipliers()` can be frontrun by the attacker with staking XVS and/or lending/borrowing transactions in order to increase the attacker's score before 'freezing' it.

If the attacker wants to keep the inflated score, no actions that update the attacker's score can be taken. However, the attacker can claim the outsized rewards earned at any time and as often as desired since `claimInterest()` does not update user scores.

If anyone has knowledge that the exploit has occurred, it is possible for any user's score in a market to be updated with a call to `accrueInterestAndUpdateScore()` , which can neutralize the attack.

The required amount of XVS staked for this exploit can be reduced by 1000 if this exploit is combined with the exploit titled "Irrevocable token holders can instantly mint a revocable token after burning and bypass the minimum XVS stake for revocable tokens".

chechu (Venus) confirmed and commented:

> Fixed here.

[H-03] Incorrect decimal usage in score calculation leads to reduced user reward earnings

Submitted by Brenzee*, also found by* Testerbot, 0xDetermination, santipu_, ast3ros, ether_sky, sces60107, pep7siup, Breeje, tapir, *0xTheC0der (*1, 2*), and* SpicyMeatball

Users earned rewards are calculated incorrectly because of the incorrect decimals value used to calculate user's `score` and markets `sumOfMembersScore` , which impacts the `delta` that is added to market's `rewardIndex` when `Prime.accrueInterest` function is called.

Proof of Concept

All users rewards are calculated with the following formula:

```
    rewards = (rewardIndex - userRewardIndex) * scoreOfUser;
```

This means that for user to earn rewards, market's `rewardIndex` needs to periodically increase.

`markets[vToken].rewardIndex` is updated (increased), when `Prime.accrueInterest` is called.

`Prime.sol:L583-L588`

```
        uint256 delta;
        if (markets[vToken].sumOfMembersScore > 0) {
            delta = ((distributionIncome * EXP_SCALE) / markets[vTok
        }

        markets[vToken].rewardIndex = markets[vToken].rewardIndex +
```

From the code snippet above it is assumed that `markets[vToken].sumOfMembersScore` is precision of 18 decimals.

To ensure that user's `score` and markets `sumOfMemberScore` are correctly calculated, in `Prime._calculateScore` function `capital` is adjusted to 18 decimals. After that `capital` is used in `Scores.calculateScore` function. Note: `capital` precision from `_capitalForScore` function is in precision of **underlying token decimals.**

`Prime.sol:660-L663`

```
        (uint256 capital, , ) = _capitalForScore(xvsBalanceForScore,
        capital = capital * (10 ** (18 - vToken.decimals()));

        return Scores.calculateScore(xvsBalanceForScore, capital, al
```

The mistake is made when `vToken.decimals()` is used instead of `vToken.underlying().decimals()`.

To prove that this is the case, here are vTokens deployed on Binance Smart chain, their decimals and underlying token decimals:

vToken	vToken decimals	Underlying token decimals
vUSDC	8	18
vETH	8	18
vBNB	8	18
vBTC	8	18
vUSDT	8	18

Since `vToken.decimals()` is used, this means the precision of `capital` is `18 + (18 - 8) = 28` decimals instead of 18 decimals, which makes the `score` calculation from `Score.calculateScore` function incorrect, since the function expects `capital` to be in precision of 18 decimals.

As a result, `delta` for market's `rewardIndex` is incorrectly calculated and it can be 0 even though it shouldn't be, which means that users will not accrue any rewards.

Update current test with correct decimals for vTokens

Developers have made a mistake when writing the tests for `Prime.sol` - in the tests they have set vToken decimals to 18 instead of 8, which makes the tests pass, but on the Binance Smart Chain all of the vToken decimals are 8.

If the decimal value of vToken is set to 8 in the tests, then the tests will fail.

Change the `vusdt`, `veth` and `vbnb` decimals to 8 and run:

```
npx hardhat test tests/hardhat/Prime/*.ts tests/hardhat/integrat
```

This will make the current tests fail.

PoC Test

Here is a test where it shows, that `rewardIndex` is still 0 after `Prime.accrueInterest` is called, even though it should be > 0.

PoC test

Recommended Mitigation Steps

Make sure that underlying token decimals are used instead of vToken decimals when calculating `capital` in `Prime._calculateScore` function.

```
        (uint256 capital, , ) = _capitalForScore(xvsBalanceForScore,
        capital = capital * (10 ** (18 - IERC20Upgradeable(_getUnder
```

chechu (Venus) confirmed via duplicate issue 588 and commented:

> Fixed. See here and here

0xDjango (Judge) commented:

> Agree with high severity as the core mechanic of interest calculation is incorrect.

Medium Risk Findings (3)

[M-01] Scores.sol: Incorrect computation of user's score when alpha is 1

Submitted by Testerbot

The formula for a user's score depends on the `xvs` staked and the `capital`. One core variable in the calculation of a user's score is `alpha` which represents the weight for `xvs` and `capital`. It has been stated in the documentation that `alpha` can range from 0 to 1 depending on what kind of incentive the protocol wants to drive (XVS Staking or supply/borrow). Please review ***Significance of α*** subsection.

When `alpha` is 1, `xvs` has all the weight when calculating a user's score, and `capital` has no weight. If we see the Cobb-Douglas function, the value of `capital` doesn't matter, it will always return 1 since `capital^0` is always 1. So, a user does not need to borrow or lend in a given market since `capital` does not have any weight in the score calculation.

The issue is an inconsistency in the implementation of the Cobb-Douglas function.

Developers have added an exception: `if (xvs == 0 || capital == 0) return 0;`
Because of this the code will always return 0 if either the `xvs` or the `capital` is zero, but we know this should not be the case when the `alpha` value is 1.

PoC

To check how it works:

In `describe('boosted yield')` add the following code:

```
    it.only("calculate score only staking", async () => {
        const xvsBalance = bigNumber18.mul(5000);
        const capital = bigNumber18.mul(0);

        await prime.updateAlpha(1, 1); // 1
        //  5000^1 * 0^0 = 5000
        // BUT IS RETURNING 0!!!
        expect((await prime.calculateScore(xvsBalance, capital)).t
    });
```

Recommended Mitigation Steps

Only return 0 when (`xvs = 0` or `capital = 0`) * and `alpha` is not 1.

[chechu (Venus) confirmed and commented]():

> Fixed [here](). (Alpha cannot be 0 or 1 anymore)

[0xDjango (Judge) commented]():

> I agree with Medium severity. In the case where Venus decides to make XVS staking the sole factor in user score, this function should calculate correctly. Also given that the documentation specifically states "The value of α is between 0-1".

[M-02] DoS and gas griefing of calls to Prime.updateScores()

Submitted by [0xDetermination](), *also found by* [SBSecurity](), *maanas (*[1](), [2]()*)*, [al88nsk](), [PwnStars](), [Pessimistic](), [dethera](), [ThreeSigma](), [0xblackskull](), [xAriextz](), [blutorque](), [neumo](), [tsvetanovv](), [sces60107](), [pina](), [oakcobalt](), [Breeje](), [ADM](), [tapir](), [said](), [debo](), [0xweb3boy](), *and* [Satyam_Sharma]()

[https://github.com/code-423n4/2023-09-venus/blob/main/contracts/Tokens/Prime/Prime.sol#L200-L230]()

[https://github.com/code-423n4/2023-09-venus/blob/main/tests/hardhat/Prime/Prime.ts#L294-L301]()

`updateScores()` is meant to be called to update the scores of many users after reward alpha is changed or reward multipliers are changed. An attacker can cause calls to `Prime.updateScores()` to out-of-gas revert, delaying score updates. Rewards will be distributed incorrectly until scores are properly updated.

Proof of Concept

`updateScores()` will run out of gas and revert if any of the `users` passed in the argument array have already been updated. This is due to the `continue` statement and the incrementing location of `i`:

```
        function updateScores(address[] memory users) external {
            if (pendingScoreUpdates == 0) revert NoScoreUpdatesRequi
            if (nextScoreUpdateRoundId == 0) revert NoScoreUpdatesRe

            for (uint256 i = 0; i < users.length; ) {
                address user = users[i];

                if (!tokens[user].exists) revert UserHasNoPrimeToker
                if (isScoreUpdated[nextScoreUpdateRoundId][user]) cc
                ...
                unchecked {
                    i++;
                }

                emit UserScoreUpdated(user);
            }
        }
```

An attacker can frontrun calls to `updateScores()` with a call to `updateScores()`, passing in a 1-member array of one of the addresses in the frontran call's argument array. The frontran call will run out of gas and revert, and only one of the users intended to be updated will actually be updated.

Test

Paste and run the below test in the 'mint and burn' scenario in Prime.ts (line 302)

```
        it("dos_updateScores", async () => {
          //setup 3 users
          await prime.issue(true, [user1.getAddress(), user2.getAddr
          await xvs.connect(user1).approve(xvsVault.address, bigNumk
          await xvsVault.connect(user1).deposit(xvs.address, 0, bigN
          await xvs.connect(user2).approve(xvsVault.address, bigNumk
          await xvsVault.connect(user2).deposit(xvs.address, 0, bigN
          await xvs.connect(user3).approve(xvsVault.address, bigNumk
          await xvsVault.connect(user3).deposit(xvs.address, 0, bigN
          //change alpha
          await prime.updateAlpha(1, 5);
          //user 1 frontruns updateScores([many users]) with updateS
          await prime.connect(user1).updateScores([user3.getAddress(
          //frontran call should revert
          await expect(prime.updateScores([user1.getAddress(), user2
        });
```

Tools Used

Hardhat

Recommended Mitigation Steps

Increment `i` such that `continue` doesn't result in infinite looping:

```
      function updateScores(address[] memory users) external {
          ...
+         for (uint256 i = 0; i < users.length; ++i} ) {
-         for (uint256 i = 0; i < users.length; ) {
              ...
-             unchecked {
-                 i++;
-             }
              ...
          }
      }
```

0xRobocop (Lookout) commented:

> Finding is correct. The function `updateScore` will unsuccessfully increment `i` and hence will not skip the user, causing the call to revert.

> Consider QA, since the function can be called again.

> Marking as primary for sponsor review anyways.

chechu (Venus) confirmed, but disagreed with severity and commented:

> Consider QA.

> `updateScores` could be invoked again.

[0xDjango (Judge) commented]():

> I agree with medium severity. Take the scenario where each user update costs 10_000 gas.

- 100 users need to be updated = `10_000 * 100` = 1_000_000 gas.
- Attacker frontruns as described above, costing themselves ~10,000 gas.
- OOG error will cause original caller to spend 1_000_000 gas.
- Original caller tries again.
- Attacker frontruns again.
- OOG error will cause original caller to spend another 1_000_000 gas.

> This griefing attack can cause a significant amount of required spending.

[chechu (Venus) commented]():

> Fixed [here]().

[M-03] The `owner` is a single point of failure and a centralization risk

Submitted by [Tera Bot]
([https://gist.github.com/code423n4/9e80eddfb29953d8b5a424084a54e4ed?permalink]()comment*id=4762845#m01-the-owner-is-a-single-point-of-failure-and-a-centralization-risk)*

Note: this finding was reported via the winning [Automated Findings report](). *It was declared out of scope for the audit, but is being included here for completeness.*

Having a single EOA as the only owner of contracts is a large centralization risk and a single point of failure. A single private key may be taken in a hack, or the sole holder of the key may become unable to retrieve the key when necessary. Consider changing to a multi-signature setup, or having a role-based authorization model.

There are 3 instances of this issue:

```
File: contracts/Tokens/Prime/PrimeLiquidityProvider.sol

118         function initializeTokens(address[] calldata tokens_

177         function setPrimeToken(address prime_) external only

216         function sweepToken(IERC20Upgradeable token_, addres
```

GitHub: [118, 177, 216]

chechu (Venus) acknowledged and commented:

> Regarding [M-03] The owner is a single point of failure and a centralization risk, the owner won't be an EOA, but that cannot be specified in the solidity code. The owner of our contracts is always the Normal Timelock contract deployed at 0x939bD8d64c0A9583A7Dcea9933f7b21697ab6396. This contract is used in the Governance process, so after a voting period of 24 hours, and an extra delay of 48 hours if the vote passed, this contract will execute the commands agreed on the Venus Improvement Proposal.

0xDjango (Judge) commented:

> Confirming medium severity for centralization risk, though Venus does has safeguards in place to ensure that decentralization is achieved through governance. From a user-perspective, the potentiality of centralization risks must be highlighted.

Low Risk and Non-Critical Issues

For this audit, 88 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Bauchibred** received the top score from the judge.

The following wardens also submitted reports: 0xprinc, 0xDetermination, DavidGiladi, josephdara, 0xTheC0der, Testerbot, 0xScourgedev, Maroutis, inzinko, ge6a, merlin, rokinot, 0xMosh, btk, bin2chen, 0xdice91, Fulum, PwnStars, gkrastenov, ThreeSigma, imare, Aymen0909, Flora, DeFiHackLabs, SPYBOY, KrisApostolov, J4X, neumo, jnforja, alexweb3, Mirror, berlin-101, deth, pina, 0xTiwa, d3e4, twicek, ast3ros, tonisives, 0xpiken, MohammedRizwan, ether_sky, pep7siup, Brenzee, Tricko, jkoppel, lsaudit, 0xfusion, glcanvas, lotux, santipu_, xAriextz, y4y, blutorque, ArmedGoose, sashik_eth, Krace, Daniel526, al88nsk, Norah, HChang26, squeaky_cactus, joaovwfreire, Breeje, oakcobalt, hals, rvierdiiev, peanuts, mahdirostami, vagrant, nobody2018, nisedo, IceBear, terrancrypt, Hama, said, n1punp, nadin, e0d1n, kutugu, seerether, 0xWaitress, 0x3b, 0xweb3boy, TangYuanShen, orion, *and* ptsanev.

[01] Prime.sol's `_claimInterest()` should apply a slippage or allow users to partially claim their interest

Medium to low impact. The `_claimInterest` function hardcodes a 0% percent slippage and does not perform a final check after attempting to release funds from various sources. If the contract's balance is still insufficient after these fund releases, it could lead to a revert on the `safeTransfer` call, preventing users from claiming their interest.

Proof of Concept

Take a look at the `_claimInterest()` function:

```
    function _claimInterest(address vToken, address user) internal r
      uint256 amount = getInterestAccrued(vToken, user);
      amount += interests[vToken][user].accrued;

      interests[vToken][user].rewardIndex = markets[vToken].rewardIr
      interests[vToken][user].accrued = 0;

      address underlying = _getUnderlying(vToken);
      IERC20Upgradeable asset = IERC20Upgradeable(underlying);

      if (amount > asset.balanceOf(address(this))) {
        address[] memory assets = new address[](1);
        assets[0] = address(asset);
        IProtocolShareReserve(protocolShareReserve).releaseFunds(com
        if (amount > asset.balanceOf(address(this))) {
          IPrimeLiquidityProvider(primeLiquidityProvider).releaseFur
          //@audit-issue M no extra check to see if the balance is s
          unreleasedPLPIncome[underlying] = 0;
        }
      }

      asset.safeTransfer(user, amount);

      emit InterestClaimed(user, vToken, amount);

      return amount;
    }
```

In the above function, the contract tries to ensure it has enough balance to pay out the user's interest in the following way:

1. If the required `amount` is greater than the contract's balance, it releases funds from `IProtocolShareReserve`.
2. If the amount is still greater, it attempts to release funds from `IPrimeLiquidityProvider`.

The potential issue arises after this. There is no final check to ensure that the `amount` is less than or equal to the contract's balance before executing `safeTransfer`, do note that a slippage of 0% has been applied.

If both releases (from `IProtocolShareReserve` and `IPrimeLiquidityProvider`) do not provide sufficient tokens, the `safeTransfer` will revert, and the user will be unable to claim their interest.

Depending on the situation this might not only lead to user frustration but also cause trust issues with the platform and would lead to user funds to be stuck s

Recommended Mitigation Steps

To address this issue, the following changes are suggested:

- Add a final balance check.
- If the balance is still insufficient, send whatever is available to the user.
- Store the deficit somewhere and inform the user that there's more to claim once the contract has sufficient funds.

A diff for the recommended changes might look like:

```
    function _claimInterest(address vToken, address user) internal r
        ...
        if (amount > asset.balanceOf(address(this))) {
            address[] memory assets = new address[](1);
            assets[0] = address(asset);
            IProtocolShareReserve(protocolShareReserve).releaseFunds
            if (amount > asset.balanceOf(address(this))) {
                IPrimeLiquidityProvider(primeLiquidityProvider).rele
                unreleasedPLPIncome[underlying] = 0;
            }
        }

+       uint256 availableBalance = asset.balanceOf(address(this));
+       uint256 transferAmount = (amount <= availableBalance) ? amou

-       asset.safeTransfer(user, amount);
+       asset.safeTransfer(user, transferAmount);

        emit InterestClaimed(user, vToken, transferAmount);

        return transferAmount;
    }
```

This ensures that a user always gets whatever is available, if the third suggestion is going to be implemented then do not forget to store the differences in a variable

> NB: Submitting as QA, due to the chances of this happening being relatively low, but leaving the final decision to judge to upgrade to Medium if they see fit.

[02] `getEffectiveDistributionSpeed()` should prevent reversion and return `0` if `accrued` is ever more than `balance`

Low impact, since this is just how to better implement `getEffectiveDistributionSpeed()` to cover more valid *edge cases*.

Proof of Concept

Take a look at `getEffectiveDistributionSpeed()` :

```
function getEffectiveDistributionSpeed(address token_) external
  uint256 distributionSpeed = tokenDistributionSpeeds[token_];
  uint256 balance = IERC20Upgradeable(token_).balanceOf(address
  uint256 accrued = tokenAmountAccrued[token_];

  if (balance - accrued > 0) {
    return distributionSpeed;
  }

  return 0;
}
```

As seen, the above function is used to get the rewards per block for a specific token; the issue is that an assumption is made that `balance >= accrued` is always true, which is not the case.

To support my argument above, note that the `PrimeLiquidityProvider.sol` also employs functions like `sweepTokens()` and `releaseFunds` . Whereas the latter makes an update to the accrual during its execution (i.e., setting the `tokenAmountAccrued[token_]` to `0`), the former does not. This can be seen below:

```
function sweepToken(IERC20Upgradeable token_, address to_, uint2
  uint256 balance = token_.balanceOf(address(this));
  if (amount_ > balance) {
    revert InsufficientBalance(amount_, balance);
  }

  emit SweepToken(address(token_), to_, amount_);

  token_.safeTransfer(to_, amount_);
}

function releaseFunds(address token_) external {
  if (msg.sender != prime) revert InvalidCaller();
  if (paused()) {
    revert FundsTransferIsPaused();
  }
```

```
        accrueTokens(token_);
        uint256 accruedAmount = tokenAmountAccrued[token_];
        tokenAmountAccrued[token_] = 0;

        emit TokenTransferredToPrime(token_, accruedAmount);

        IERC20Upgradeable(token_).safeTransfer(prime, accruedAmount);
    }
```

Now, in a case where a portion of a token has been swept and the accrual is now more than the balance of said token, the `getEffectiveDistributionSpeed()` function reverts due to an underflow from the check here:

```
        uint256 distributionSpeed = tokenDistributionSpeeds[token_];
        uint256 balance = IERC20Upgradeable(token_).balanceOf(address
        uint256 accrued = tokenAmountAccrued[token_];

        if (balance - accrued > 0)
```

Recommended Mitigation Steps

Make an addition to cover up the edge case in the `getEffectiveDistributionSpeed()` function, i.e., if accrued is already more than the balance then the distribution speed should be the same as if the difference is 0.

An approach to the recommended fix can be seen from the diff below:

```diff
      function getEffectiveDistributionSpeed(address token_) exter
          uint256 distributionSpeed = tokenDistributionSpeeds[toke
          uint256 balance = IERC20Upgradeable(token_).balanceOf(ac
          uint256 accrued = tokenAmountAccrued[token_];
-         if (balance - accrued > 0) {
+         if (balance > accrued) {
              return distributionSpeed;
          }

          return 0;
      }
```

[03] Inconsistency in parameter validations between sister functions

Impact is low/informational.

The inconsistency in validation checks between similar functions can lead to unexpected behavior.

Proof of Concept

Take a look at `updateAlpha()` :

▶ Details

In the `updateAlpha` function, there is a check `_checkAlphaArguments(_alphaNumerator, _alphaDenominator)` which ensures the correctness of the arguments provided. This establishes a certain standard of validation for updating alpha values.

```
    function _checkAlphaArguments(uint128 _alphaNumerator, uint128 _
      if (_alphaDenominator == 0 || _alphaNumerator > _alphaDenomina
        revert InvalidAlphaArguments();
      }
    }
```

However, in the `updateMultipliers` function, there is no equivalent validation function like `_checkAlphaArguments` . This discrepancy implies that the two functions, which are logically do not maintain a consistent standard of input validation.

Recommended Mitigation Steps

Consider introducing a validation function (e.g., `_checkMultipliersArguments`) that will validate the parameters `supplyMultiplier` and `borrowMultiplier` .

- Incorporate this validation function into the `updateMultipliers` function, similar to how `_checkAlphaArguments` is used in the `updateAlpha` function.

- Make sure that the validation function checks for logical constraints, edge cases, or any other relevant criteria for these parameters.

A diff for the recommended changes might appear as:

```
+ function _checkMultipliersArguments(uint256 supplyMultiplier,
+     // Implement the validation logic here
+     ...
+ }

function updateMultipliers(address market, uint256 supplyMultipl
    ...
+   _checkMultipliersArguments(supplyMultiplier, borrowMultiplie
    ...
    if (!markets[market].exists) revert MarketNotSupported();
    ...
    emit MultiplierUpdated(
        ...
    );
    markets[market].supplyMultiplier = supplyMultiplier;
    markets[market].borrowMultiplier = borrowMultiplier;

    _startScoreUpdateRound();
}
```

[04] The `lastAccruedBlock` naming needs to be refactored

Low impact, confusing comment/docs since the `lastAccruedBlock` is either wrongly implemented or wrongly named.

Proof of Concept

Take a look at PrimeLiquidityProvider.sol#L22-L24:

```
    /// @notice The rate at which token is distributed to the Pr
    mapping(address => uint256) public lastAccruedBlock;
```

As seen, `lastAccruedBlock` currently claims to be the rate at which a token is distributed to the prime contract, but this is wrong since this mapping actuallly holds value for the last block an accrual was made for an address.

Recommended Mitigation Steps

Fix the comments around the `lastAccruedBlock` mapping or change the name itself.

[05] More sanity checks should be applied to `calculateScore()`

Low impact, since the chances of passing a valuse above type(int256).max is relatively slim.

Proof of Concept

Take a look at `calculateScore()`:

▶ Details

Do note that the function is used to calculate a membership score given some amount of `xvs` and `capital`, issue is that both alpha numerators and denoimantors are passed in as `uint256` values and then while getting the `exponentiation` value these values are converted to `int256` which would cause a revert.

Recommended Mitigation Steps

Better code structure could be applied, from the get go the values passed in for numerators and denominators could be checked to ensure that they are not above the max value of int256.

[06] The `getPendingInterests()` should be better implemented to handle cases where a market is unavailable

Low impact. If one market goes down or becomes unresponsive, users won't be able to retrieve their pending interests for any of the subsequent markets. This can lead to misinformation and might affect user decisions.

Proof of Concept

Take a look at `getPendingInterests()`:

```
function getPendingInterests(address user) external returns (Per
  address[] storage _allMarkets = allMarkets;
  PendingInterest[] memory pendingInterests = new PendingIntere
  //@audit-ok this function currently does not take into account

  for (uint256 i = 0; i < _allMarkets.length; ) {
    address market = _allMarkets[i];
    uint256 interestAccrued = getInterestAccrued(market, user);
    uint256 accrued = interests[market][user].accrued;

    pendingInterests[i] = PendingInterest({ market: IVToken(mar

    unchecked {
      i++;
    }
  }

  return pendingInterests;
}
```

As seen, there's a loop that goes through all available markets to fetch the boosted pending interests for a user. However, if any of these calls (like `getInterestAccrued`) fail due to market unavailability or any other issue, the loop will be interrupted.

Recommended Mitigation Steps

To ensure the function's resilience and provide accurate information even if some markets are down, wrap the calls inside the loop in a `try/catch` block. If a call fails, log an event indicating the market that caused the failure:

```
event MarketCallFailed(address indexed market);

for (uint256 i = 0; i < _allMarkets.length; ) {
    address market = _allMarkets[i];

    try {
        uint256 interestAccrued = getInterestAccrued(market, use
        uint256 accrued = interests[market][user].accrued;

        pendingInterests[i] = PendingInterest({
            market: IVToken(market).underlying(),
            amount: interestAccrued + accrued
        });
    } catch {
        emit MarketCallFailed(market);
    }

    unchecked {
        i++;
    }
}
```

By incorporating this change, even if a market fails, the loop won’t break, and the function will continue to retrieve the pending interests for the remaining markets. The emitted event will notify off-chain systems or users about the problematic market.

[07] Missing zero/isContract checks in important functions

Low impact, since this requires a bit of an admin error, but some functions which could be more secure using the `_ensureZeroAddress()` currently do not implement this and could lead to issues.

Proof of Concept

Using the `PrimeLiquidityProvider.sol` in scope as acase study. Below is the implementation of `_ensureZeroAddress()`

```
    function _ensureZeroAddress(address address_) internal pure {
      if (address_ == address(0)) {
        revert InvalidArguments();
      }
    }
```

As seen the above is used within protocol as a modifier/function to revert whenever addresses are being passed, this can be seen to be implemented in the `setPrime()` function and others, but that’s not always the case and in some instances addresses are not ensured to not be zero.

Additionally, as a security measure an `isContract()` function could be added and used to check for instances where the provided addresses must be valid contracts with code.

Recommended Mitigation Steps

Make use of `_ensureZeroAddress()` in all instances where addresses could be passed.

If the `isContract()` is going to be implemented then the functionality to be added could look something like this:

```
function isContract(address addr) internal view returns (bool) {
  uint256 size;
  assembly {
    size := extcodesize(addr)
  }
  return size > 0;
}
```

And then, this could be applied to instances where addreses must have byte code.

[08] The `uintDiv()` function from `FixedMath.sol` should protect against division by `0`

Low impact, just info on how to prevent panic errors cause by `division by zero`

Proof of Concept

Take a look at `uintDiv()`:

```
function uintDiv(uint256 u, int256 f) internal pure returns (uin
  if (f < 0) revert InvalidFixedPoint(); //@audit
  // multiply `u` by FIXED_1 to cancel out the built-in FIXED_1
  return uint256((u.toInt256() * FixedMath0x.FIXED_1) / f);
}
```

As seen, this function is used to divide an unsigned `int` in this case `u` by a fixed number `f`, but currently the only check applied to the `uintDiv()` function in regards to the divisor `f` is: `if (f < 0) revert InvalidFixedPoint();` The above means that a value of `0` would get accepted for `f` which would result in a panic error.

Recommended Mitigation Steps

Make changes to the `uintDiv()` function

```
        function uintDiv(uint256 u, int256 f) internal pure returns
    -         if (f < 0) revert InvalidFixedPoint();
    +         if (f <= 0) revert InvalidFixedPoint();
            // multiply `u` by FIXED_1 to cancel out the built-in FI
            return uint256((u.toInt256() * FixedMath0x.FIXED_1) / f)
        }
```

[09] Fix typo in docs

Low impact, info on how to have a better code documentation.

Proof of Concept

Take a look at the Prime token's [ReadMe]

Line 296 states:

```
    The OpenZeppelin Plausable contract is used. Only the `claimInte
```

As seen an evident typo has been missed.

Recommended Mitigation Steps

Change the line, from:

```
    The OpenZeppelin Plausable contract is used. Only the `claimInte
```

To:

```
    The OpenZeppelin Pausable contract is used. Only the `claimInter
```

[0xDjango (Judge) commented](#):

> Agree with all QA points, though a couple require design changes that may outweigh the benefit to the protocol from an added risk standpoint.

[chechu (Venus) commented](#):

1. The warden doesn't explain how Prime can get from the sources less funds than needed. Accounting avoids that scenario. Fee-on-transfer tokens could generate this issue, but they shouldn't be added as Prime market. When claimInterest() is called we transfer tokens from PrimeLiquidityProvider to Prime contract when there is insufficient funds. So users can always claim full rewards.
2. Fixed [here](#).
3. Acknowledged.
4. Fixed [here](#).
5. Acknowledged.
6. Acknowledged.
7. Partially fixed [here](#).
8. Acknowledged.
9. Fixed [here](#).

Gas Optimizations

For this audit, 11 reports were submitted by wardens detailing gas optimizations. The [report highlighted below](#) by **DavidGiladi** received the top score from the judge.

The following wardens also submitted reports: 0xhacksmithh, 0xprinc, lsaudit, pavankv, pontifex, hihen, jkoppel, oakcobalt, 0x3b, *and* 0xWaitress.

[G-01] Avoid unnecessary storage updates

- Severity: Gas Optimization
- Confidence: High
- Total Gas Saved: 2400

Note

I reported only on three issues that were missing in the winning bot.

Description

Avoid updating storage when the value hasn't changed. If the old value is equal to the new value, not re-storing the value will avoid a `SSTORE` operation (costing 2900 gas), potentially at the expense of a `SLOAD` operation (2100 gas) or a `WARMACCESS` operation (100 gas).

There are 3 instances of this issue:

[G-02] Multiplication and Division by 2 Should use in Bit Shifting

- Severity: Gas Optimization
- Confidence: High
- Total Gas Saved: 100

Description

The expressions 'x * 2' and 'x / 2' can be optimized for gas efficiency by utilizing bitwise operations. In Solidity, you can achieve the same results by using bitwise left shift (x << 1) for multiplication and bitwise right shift (x >> 1) for division.

Using bitwise shift operations (SHL and SHR) instead of multiplication (MUL) and division (DIV) opcodes can lead to significant gas savings. The MUL and DIV opcodes cost 5 gas, while the SHL and SHR opcodes incur a lower cost of only 3 gas.

By leveraging these more efficient bitwise operations, you can reduce the gas consumption of your smart contracts and enhance their overall performance.

There are 5 instances of this issue:

[G-03] Modulus operations that could be unchecked

- Severity: Gas Optimization
- Confidence: High
- Total Gas Saved: 85

Description

Modulus operations should be unchecked to save gas since they cannot overflow or underflow. Execution of modulus operations outside `unchecked` blocks adds nothing but overhead. Saves about 30 gas.

There is 1 instance of this issue:

[G-04] Short-circuit rules can be used to optimize some gas usage

- Severity: Gas Optimization
- Confidence: Medium
- Total Gas Saved: 6300

Description

Some conditions may be reordered to save an SLOAD (2100 gas), as we avoid reading state variables when the first part of the condition fails (with &&), or succeeds (with ||). For instance, consider a scenario where you have a `stateVariable` (a variable stored in contract storage) and a `localVariable` (a variable in memory).

If you have a condition like `stateVariable > 0 && localVariable > 0`, if `localVariable > 0` is false, the Solidity runtime will still execute `stateVariable > 0`, which costs an SLOAD operation (2100 gas). However, if you reorder the condition to `localVariable > 0 && stateVariable > 0`, the `stateVariable > 0` check won't happen if `localVariable > 0` is false, saving you the SLOAD gas cost.

Similarly, for the `||` operator, if you have a condition like `stateVariable > 0 || localVariable > 0`, and `stateVariable > 0` is true, the Solidity runtime will still execute `localVariable > 0`. But if you reorder the condition to `localVariable > 0 || stateVariable > 0`, and `localVariable > 0` is true, the `stateVariable > 0` check won't happen, again saving you the SLOAD gas cost.

This detector checks for such conditions in the contract and reports if any condition could be optimized by taking advantage of the short-circuiting behavior of `&&` and `||`.

There are 3 instances of this issue:

[G-05] Unnecessary Casting of Variables

- Severity: Gas Optimization
- Confidence: High

Description

This detector scans for instances where a variable is casted to its own type. This is unnecessary and can be safely removed to improve code readability.

There is 1 instance of this issue:

[G-06] Unused Named Return Variables

- Severity: Gas Optimization
- Confidence: High

Description

Named return variables allow for clear and explicit naming of values to be returned from a function. However, when these variables are unused, it can lead to confusion and make the code less maintainable.

There are 5 instances of this issue:

[G-07] Inefficient Parameter Storage

- Severity: Gas Optimization
- Confidence: Medium
- Total Gas Saved: 50

Description

When passing function parameters, using the `calldata` area instead of `memory` can improve gas efficiency. Calldata is a read-only area where function arguments and external function calls' parameters are stored.

By using `calldata` for function parameters, you avoid unnecessary gas costs associated with copying data from `calldata` to memory. This is particularly beneficial when the parameter is read-only and doesn't require modification within the contract.

Using `calldata` for function parameters can help optimize gas usage, especially when making external function calls or when the parameter values are provided externally and don't need to be stored persistently within the contract.

There is 1 instance of this issue:
chechu (Venus) commented:

1. Acknowledged. It’s an admin function and will be called only if values are different and needs an update.
2. Acknowledged.
3. Acknowledged.
4. Fixed here.
5. Acknowledged.
6. Fixed here.
7. Fixed here.

Audit Analysis

For this audit, 14 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **3agle** received the top score from the judge.

The following wardens also submitted reports: J4X, oakcobalt, 0x3b, Breeje, radev_sw, Bauchibred, 0xDetermination, hunter_w3b, kaveyjoe, ArmedGoose, versiyonbir, jamshed, *and* 0xweb3boy.

Index

Codebase quality

- The codebase exhibited remarkable quality. It has undergone multiple audits, ensuring the robustness of all major functionalities.

Strengths

- Comprehensive unit tests
- Utilization of Natspec
- Adoption of governance for major changes, reducing centaralization risks.

Weaknesses

- Documentation had room for improvement.

Mechanism Review

Architecture & Working

- Following are the main components of the Venus Prime system:

    1. **Prime.sol:**

        - *Description*: Prime.sol serves as the central contract responsible for distributing both revocable and irrevocable tokens to stakers.
        - *Functionality*: This contract plays a pivotal role in issuing Prime tokens to eligible stakers, enabling them to participate in the Venus Prime program and earn rewards.

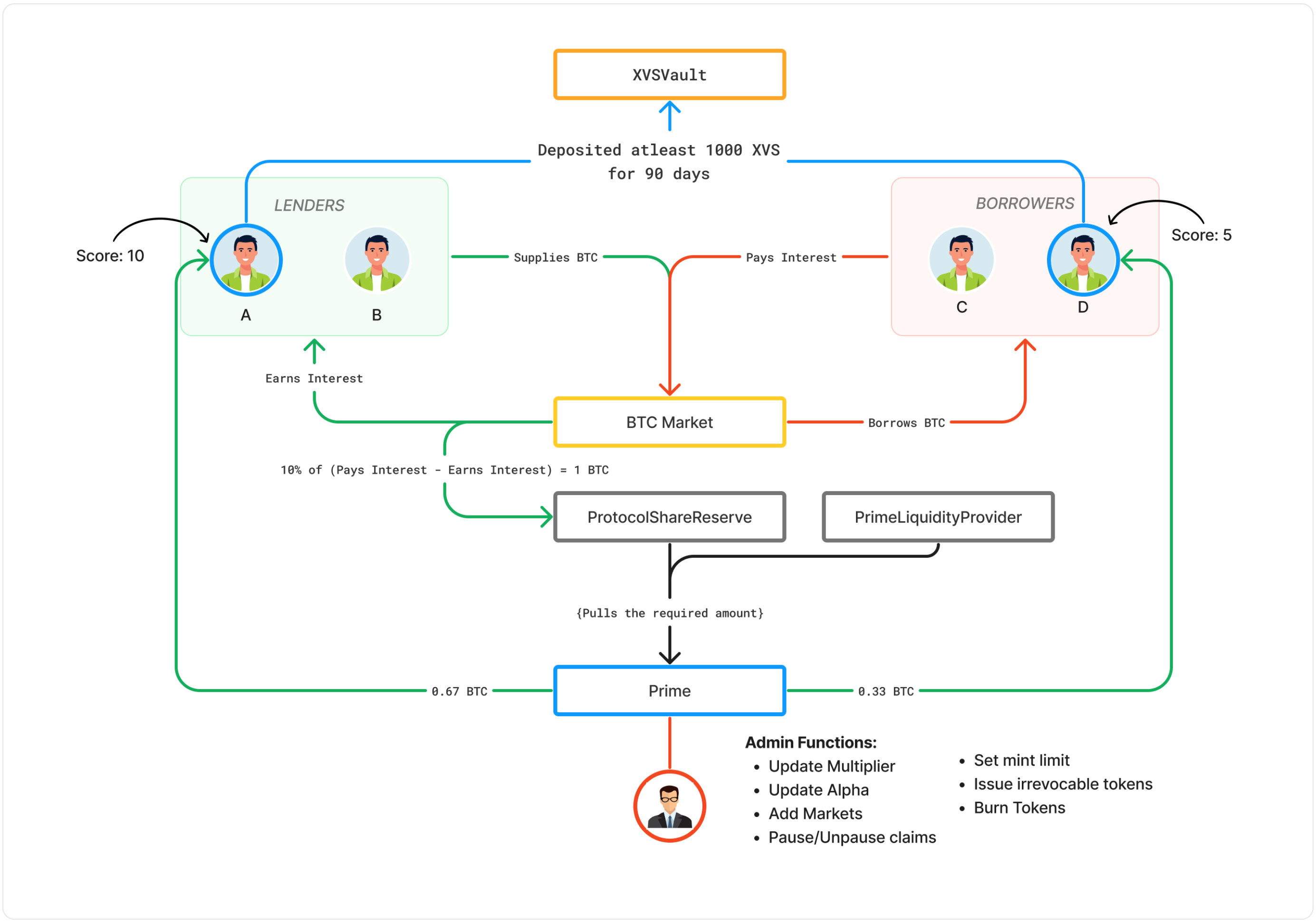
    2. **ProtocolLiquidityProvider:**

        - *Description*: The ProtocolLiquidityProvider contract holds a reserve of funds that are designated for gradual allocation to the Prime contract over a predefined time period.

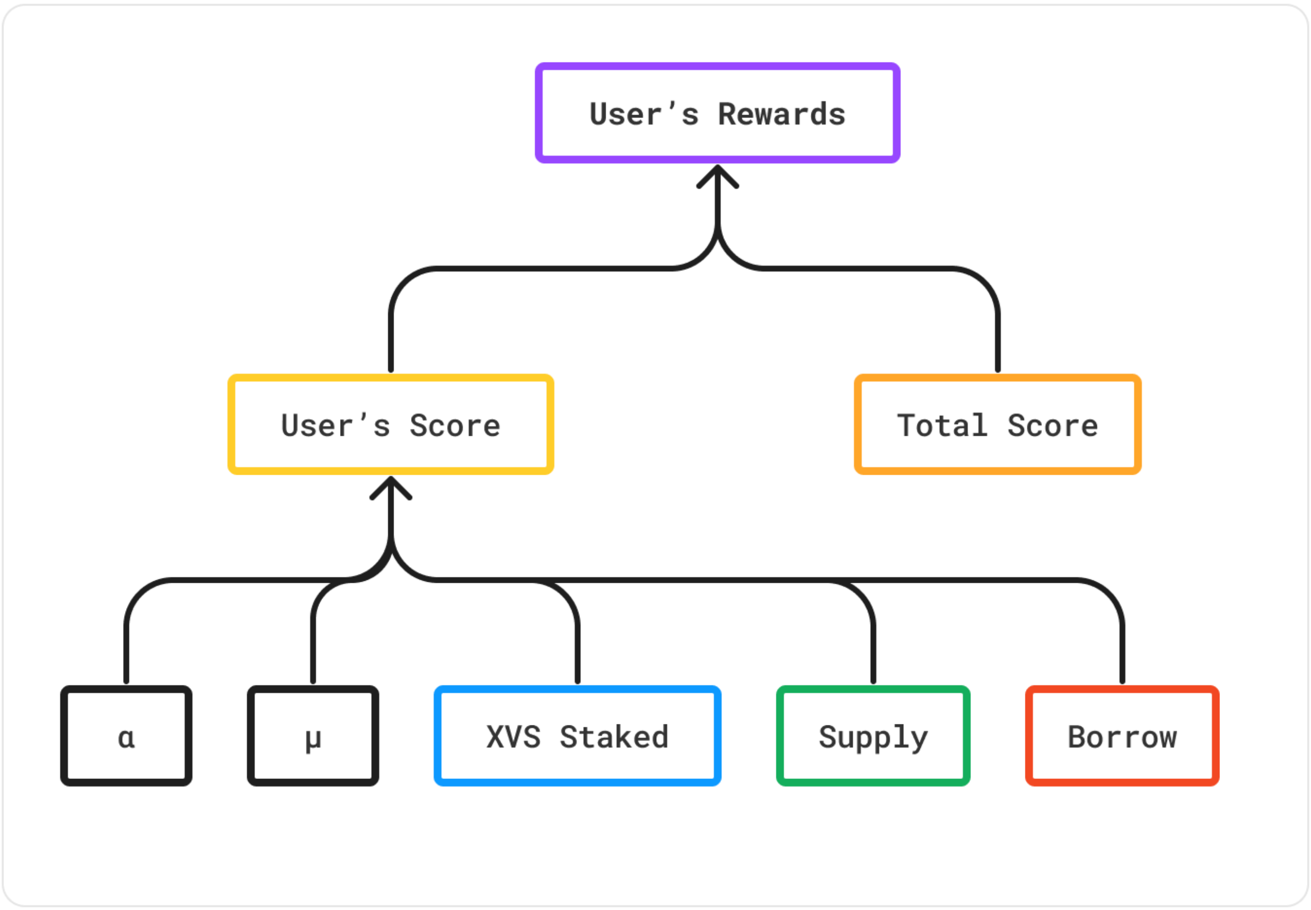
- *Functionality*: This component ensures a consistent and planned provision of tokens to the Prime contract, guaranteeing a continuous supply of rewards for Prime token holders.

3. **ProtocolShareReserve (OOS):**

    - *Description*: The ProtocolShareReserve acts as the repository for interest earnings generated across all prime markets within the Venus Protocol.
    - *Functionality*: It collects the interest earnings and acts as a centralized pool from which the Prime contract draws the necessary funds. These funds are then distributed as rewards to Prime token holders, contributing to the sustainability of the Venus Prime program.



Rewarding Mechanism



1. Initialization (Block 0):
    - Variables are set to default values.
2. Income and Scoring (Blocks 1-10):
    - Over 10 blocks, 20 USDT income is distributed.
    - User scores total 50, allocating 0.4 USDT per score point.
    - `markets[vToken].rewardIndex` increases by +0.4.
3. Income and Scoring (Blocks 11-30):
    - Next 20 blocks distribute 10 USDT.
    - User scores remain constant.
    - `markets[vToken].rewardIndex` increases by +0.2.
4. User Claims a Prime Token (Block 35):
    - User claims a Prime token.

- `markets[vToken].rewardIndex` updated via `accrueInterest`.
- Over 5 blocks, 5 USDT income with no score change.
- `markets[vToken].rewardIndex` increases by +0.1.
- `interests[market][account].rewardIndex` set to `markets[market].rewardIndex`.

5. User Claims Interest (Block 50):

    - User claims accrued interests.
    - `markets[vToken].rewardIndex` updated via `accrueInterest`.
    - Over 15 blocks, 120 USDT income with score change (sum of scores becomes 60).
    - `markets[vToken].rewardIndex` increases by +2.

6. Interest Calculation for the User:

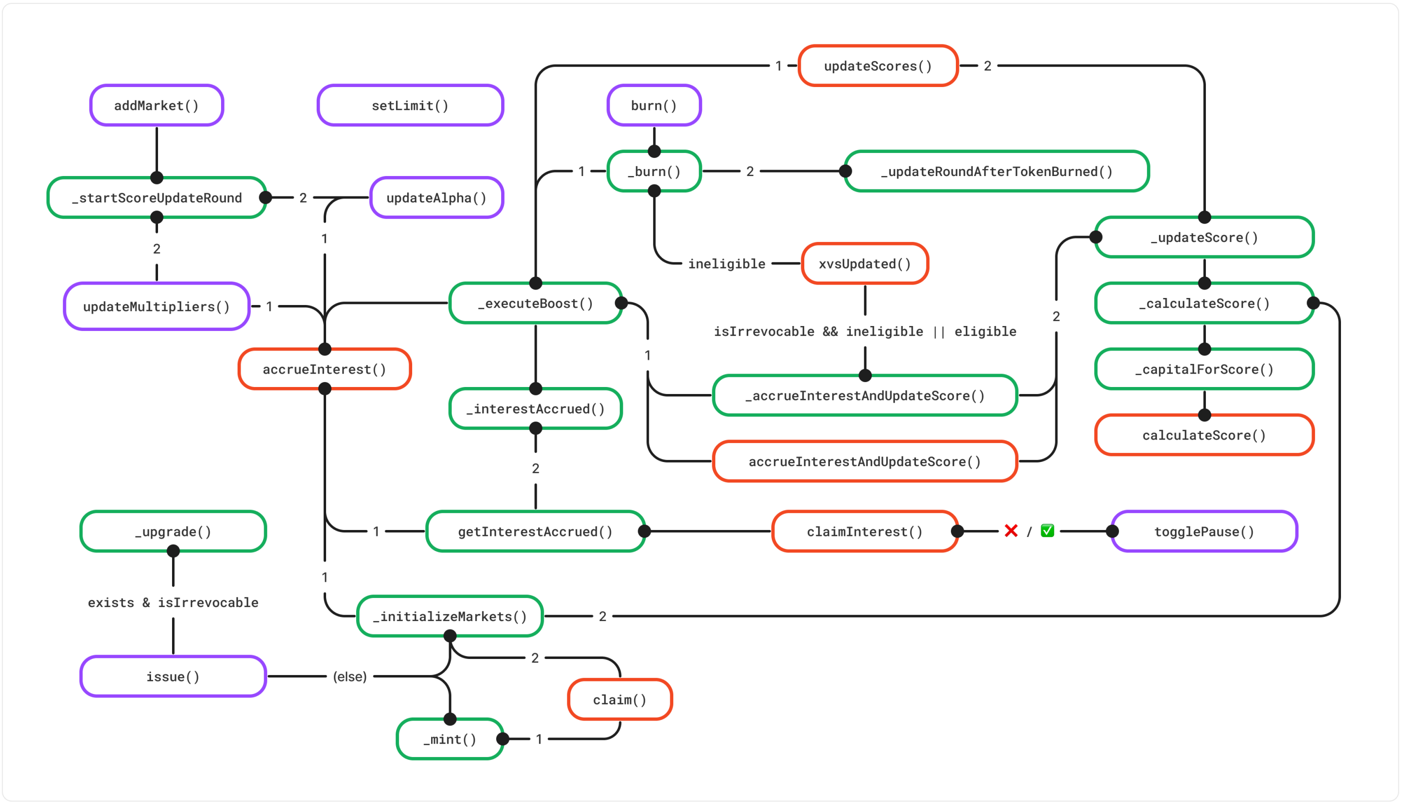
    - User's interest calculated using `_interestAccrued`.
    - The difference between `markets[vToken].rewardIndex` (2.7) and `interests[market][account].rewardIndex` (0.7) is 2 USDT per score point.
    - Multiplied by user's score (10) equals 20 USDT accrued interest.

7. Updating User's Reward Index:

    - `interests[vToken][user].rewardIndex` set to the new `markets[vToken].rewardIndex`.

Prime

Prime

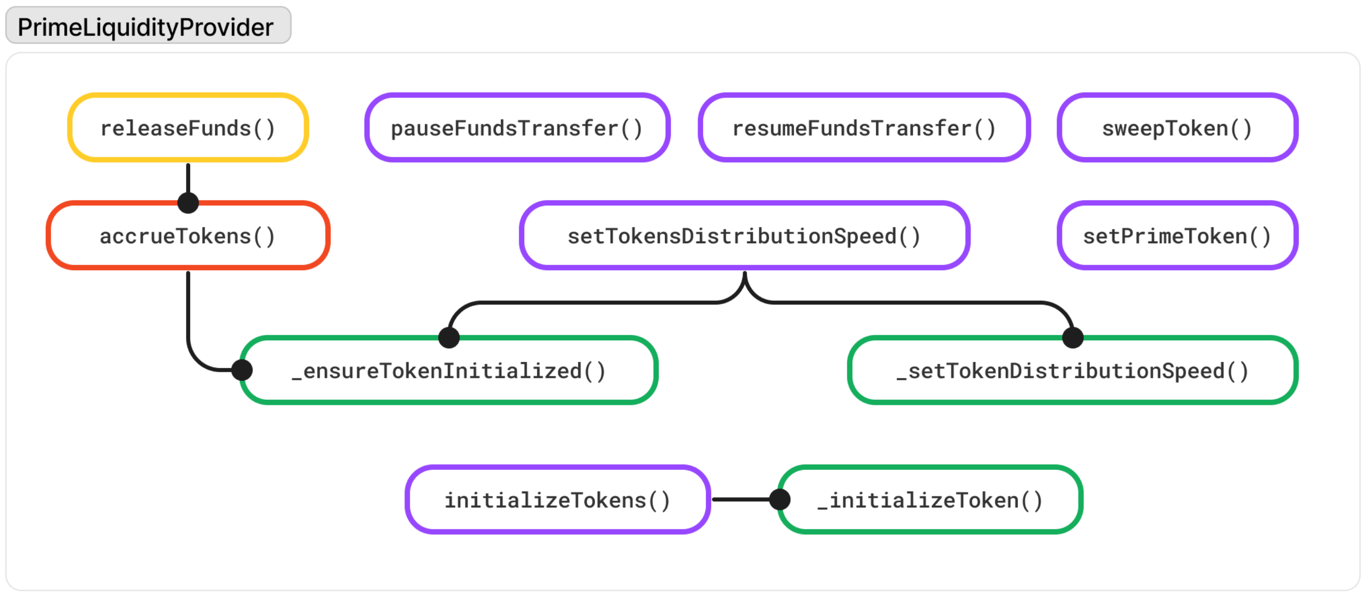


- `Prime.sol` is a critical component of Venus Protocol, offering users the opportunity to earn rewards generated from specific markets within the protocol. Here's how it works:
  - To be eligible for a Prime token, regular users must stake a minimum of 1,000 XVS for a continuous period of 90 days.
  - Once this condition is met, users can claim their Prime token, signaling the beginning of rewards accrual.
  - Prime token holders have the flexibility to claim their accrued rewards at their convenience.

Key Features and Functions in `Prime.sol`:

- Utilizes the Cobb-Douglas function to calculate user rewards.
- Offers two types of Prime Tokens:
  - Revocable: Can be minted after staking a minimum of 1,000 XVS for 90 days but can be burnt if the stake falls below 1,000 XVS.
  - Irrevocable (Phase 2) - Specifics for this token type will be detailed in Phase 2 of the Venus Prime program.

PrimeLiquidityProvider



- `PrimeLiquidityProvider.sol` serves as the second source of tokens for the Prime program, complementing the tokens generated by Venus markets. Key details include:
    - This contract, `PrimeLiquidityProvider`, allocates a predetermined quantity of tokens to Prime holders in a uniform manner over a defined period.

Key Features and Functions in `PrimeLiquidityProvider.sol`:

- Ability to add new tokens for rewards accrual.
- Configurable token distribution speed.
- Safeguard mechanisms for accidentally sent tokens.
- Ability to pause and resume token transfers, ensuring control and stability within the program.

Architecture Recommendations

- **Staking and Earning Venus Prime Tokens:** These are essential parts of the Venus Prime Yield Protocol. Let’s break them down and talk about ways to make them even better:
- **Staking as it is now:**

    - Right now, if you want to earn a Prime Token, you have to lock up at least 1,000 XVS for 90 days. This Prime Token represents the XVS you've staked and comes with some perks. But if you take out some XVS and drop below 1,000, you'll lose your Prime Token.
- **Ways to Make Staking Better:**
    - **Different Staking Levels**: Instead of one fixed requirement, we could have different levels for staking. This way, even if you have fewer tokens, you can join in. Different levels might offer different bonuses to encourage more staking.
    - **Flexible Staking Time**: We could let you choose how long you want to stake your XVS, depending on what suits you. Staking for longer could mean more rewards, which could motivate people to keep their XVS locked up longer.
    - **Smart Staking Rules**: The rules for how much you need to stake could change depending on how the market is doing and how much XVS is worth. This could help keep things steady and reliable.
- **What You Can Do with Venus Prime Tokens:**
    - **More Ways to Use Them**: We could give you more things to do with your Prime Tokens, so they're even more useful in the Venus ecosystem. That could encourage more people to join in.
    - These tokens are like keys to the Venus ecosystem. You can use them to vote on changes, earn rewards, and access different parts of the system.
- These changes could make staking easier and more rewarding, and make Venus Prime Tokens more valuable. It's all about making the system better for everyone.

Centralization Risks

- **Smart Contract Centralization**: The protocol is based on smart contracts, which are autonomous and decentralized by nature. However, the developers of the protocol have the ability to update or modify these contracts, introducing a potential point of centralization. If these powers are abused or misused, it could lead to centralization risks

- **Governance Centralization:** The governance of Venus Prime is controlled by XVS token holders. If a small group of individuals or entities comes to own a large portion of XVS tokens, they could potentially control the governance of the protocol, leading to centralization. This can include making decisions that favor them at the expense of other users

Systemic Risks

1. **Governance Risk:** The protocol faced a hacking attempt where the attacker tried to gain control of the protocol by bribery (VIP42). Although the attempt was thwarted, it highlights the risk of governance attacks
2. **Risk Fund Adequacy:** Venus has a risk fund established to address potential shortfalls in the protocol, particularly in situations of ineffective or delayed liquidations. However, if the fund is not adequate to cover a major event, it could potentially result in a systemic risk
3. **Price Oracle Resilience:** Venus V4 introduces resilient price feeds. If these feeds fail or provide inaccurate data, it could potentially destabilize the system. The protocol's resilience to such an event is yet to be tested
4. **Dependence on XVS Staking:** Venus Prime requires users to stake at least 1,000 XVS for 90 days to mint their Prime Token. If a user decides to withdraw XVS and their balance falls below 1,000, their Prime Token will be automatically revoked. This introduces a risk if there's a significant drop in the value of XVS or if a large number of users decide to unstake and sell their XVS simultaneously

Key Takeaways

1. **Dual Token System:** The protocol introduces a bifurcated token system, comprising revocable and irrevocable Prime tokens. Each token variant carries its unique rules and benefits, offering users a versatile array of choices.
2. **Sustainable Rewards:** Diverging from conventional incentive models dependent on external sources, Venus Prime generates its rewards intrinsically within the protocol. This inherent mechanism not only fosters sustainability but also augments the potential for long-term growth.
3. **Long-Term Commitment:** Users are incentivized to uphold a commitment to the protocol by staking XVS tokens for a minimum duration of 90 days. This prolonged engagement serves the dual purpose of fostering dedication and dissuading premature withdrawals.

4. **Complex Reward Calculation:** Venus Prime employs a sophisticated reward calculation formula known as the Cobb-Douglas function to ascertain user rewards. This intricacy, while daunting on the surface, is intricately designed to uphold principles of equity and precision in reward distribution.

Time spent:

45 hours

Disclosures

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

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit 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