Rubicon contest
Findings & Analysis Report

2022-08-01

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 Rubicon smart contract system written in Solidity. The audit contest took place between May 23—May 28, 2022.

Wardens

110 Wardens contributed reports to the Rubicon contest:

1. WatchPug (jtp and ming)
2. cccz
3. 0x1f8b
4. berndartmueller
5. IllIllI
6. PP1004 (ngfam and TrungOre)
7. xiaoming90
8. Ruhum
9. shenwilly
10. pedroais
11. pauliax
12. kenzo
13. hansfriese
14. GimelSec (rayn and sces60107)
15. 0xsomeone
16. sseefried
17. blackscale
18. sashik_eth
19. csanuragjain
20. unforgiven
21. hubble (ksk2345 and shri4net)
22. dirk_y
23. Bahurum
24. 0x52
25. throttle

26. broccolirob
27. 0x1337
28. eccentricexit
29. CertoraInc (egjlmn1, OriDabush, ItayG, and shakedwinder)
30. Dravee
31. cryptphi
32. fatherOfBlocks
33. oyc_109
34. MiloTruck
35. dipp
36. MaratCerby
37. kebabsec (okkothejawa and FlameHorizon)
38. rotcivegaf
39. minhquanym
40. SmartSek (0xDjango and hake)
41. reassor
42. camden
43. joestakey
44. sorrynotsorry
45. 0xNoah
46. blockdev
47. gzeon
48. ilan
49. Chom
50. Kumpa
51. horsefacts
52. 0xDjango
53. Hawkeye (0xwags and 0xmint)
54. defsec

55. Waze
56. 0xNazgul
57. c3phas
58. ellahi
59. _Adam
60. catchup
61. 0x4non
62. ElKu
63. simon135
64. FSchmoede
65. Funen
66. Kaiziron
67. delfin454000
68. 0xf15ers (remora and twojoy)
69. Metatron
70. UnusualTurtle
71. sach1r0
72. Picodes
73. asutorufos
74. JC
75. VAD37
76. StErMi
77. AlleyCat
78. JMukesh
79. TerrierLover
80. BouSalman
81. ACai
82. 0xKitsune
83. UVvirus

84. parashar
85. 0xkatana
86. z3s
87. Tomio
88. antonttc
89. DavidGialdi
90. rfa
91. Randyyy
92. samruna
93. Fitraldys
94. RoiEvenHaim
95. kenta
96. peritoflores
97. aez121
98. jayjonah8
99. Deivitto

This contest was judged by hickuphh3.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 44 unique vulnerabilities. Of these vulnerabilities, 10 received a risk rating in the category of HIGH severity and 34 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 72 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 57 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 Rubicon contest repository, and is composed of 6 smart contracts written in the Solidity programming language and includes 3,351 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.

High Risk Findings (10)

[H-01] RubiconRouter: Offers created through offerWithETH() can be cancelled by anyone

Submitted by cccz, also found by kenzo, 0x1f8b, IllIIl, and pedroais

When a user creates an offer through the offerWithETH function of the RubiconRouter contract, the offer function of the RubiconMarket contract is called, and the RubiconRouter contract address is set to offer.owner in the offer function.

This means that anyone can call the cancelForETH function of the RubiconRouter contract to cancel the offer and get the ether.

Proof of Concept

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L383-L409

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L440-L452

Recommended Mitigation Steps

Set the owner of `offer_id` to msg.sender in `offerWithETH` function and check it in cancelForETH function.

bghughes (Rubicon) confirmed

[H-02] RubiconRouter: Offers created through offerForETH cannot be cancelled

Submitted by cccz

When a user creates an offer through the offerForETH function of the RubiconRouter contract, the offer function of the RubiconMarket contract is called, and the RubiconRouter contract address is set to offer.owner in the offer function.

But the RubiconRouter contract does not implement a function to cancel this offer. This means that if no one accepts the offer, the user's tokens will be locked in the contract.

Proof of Concept

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L412-L437

Recommended Mitigation Steps

Implement `cancelForERC` function to cancel this offer. And set the owner of offer_id to msg.sender in offerForETH function and check it in cancelForERC function

bghughes (Rubicon) marked as duplicate:

> Duplicate of #17.

HickupHH3 (judge) commented:

> Not a duplicate. Referring to separate lacking functionality of cancellation of ERC20 -> WETH offers (eg. a `cancelWithETH` function).

[H-03] Attacker could steal almost all the bonus tokens in BathBuddy Vesting Wallet

*Submitted by xiaoming90, also found by 0x52, PP1004, shenwilly, and sashik*eth_

BathBuddy is a Vesting Wallet that payout withdrawers any `bonusTokens` they may have accrued while staking in the Bath Token (e.g. network incentives/governance tokens).

BathBuddy Vesting Wallet releases a user their relative share of the pool's total vested bonus token during the withdraw call on BathToken.sol. This vesting occurs linearly over Unix time.

It was observed that an attacker could steal almost all the `bonusTokens` in the BathBuddy Vesting Wallet.

Proof of Concept

The root cause of this issue is that the amount of `bonusTokens` that a user is entitled to is based on their relative share of the pool's total vested bonus token at the point of the withdraw call. It is calculated based on the user's "spot" share in the pool.

Thus, it is possible for an attacker to deposit large amount of tokens into a BathToken Pool to gain significant share of the pool (e.g. 95%), and then withdraw the all the shares immediately. The withdraw call will trigger the `BathToken.distributeBonusTokenRewards` , and since attacker holds overwhelming amount of share in the pool, they will receive almost all the `bonusToken` in the BathBuddy Vesting wallet, leaving behind dust amount of `bonusToken` in the wallet. This could be perform in an atomic transaction and attacker can leverage on flash-loan to fund this attack.

The following shows an example of this issue:

1. A sponsor sent 1000 DAI to the BathBuddy Vesting Wallet to be used as `bonusTokens` for bathWETH pool. The vesting duration is 4 weeks.
2. Alice and Bob deposited 50 WETH and 50 WETH respectively. The total underlying asset of bathWETH is 100 WETH after depositing. Each of them hold 50% of the shares in the pool.
3. Fast forward to the last hour of the vesting period, most of the `bonusToken` have been vested and ready for the recipients to claim. In this example, estimate 998 DAI are ready to be claimed at the final hour.
4. Since Alice has 50% stake in the pool, she should have accured close to 449 DAI at this point. If she decided to withdraw all her bathWETH LP tokens at this point, she would receive close to 449 DAI as `bonusTokens`. But she choose not to withdraw yet.
5. Unfortunately, an attacker performed a flash-loan to borrow 8500 WETH, and deposit large amount of WETH into the bathWETH gain significant share of the pool, and then withdraw the all the shares immediately.
6. Since attacker hold the an overwhelming amount of shares in the pool, they will receive almost all the `bonusToken` (around 997 DAI) in the BathBuddy Vesting wallet, leaving behind dust amount of `bonusToken` in the wallet.
7. At this point, Alice decided to withdraw all her bathWETH LP token. She only received dust amount of 0.7 DAI as `bonusTokens`

The following code shows that the amount of `bonusTokens` a user is entitled is based on the user's current share in the pool - `amount = releasable * sharesWithdrawn/initialTotalSupply`.

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L87

```
    /// @inheritdoc IBathBuddy
    /// @dev Added and modified release function. Should be the only
    function release(
        IERC20 token,
        address recipient,
        uint256 sharesWithdrawn,
        uint256 initialTotalSupply,
        uint256 poolFee
    ) external override {
        require(
            msg.sender == beneficiary,
            "Caller is not the Bath Token beneficiary of these rewar
        );
        uint256 releasable = vestedAmount(
            address(token),
            uint64(block.timestamp)
        ) - released(address(token));
        if (releasable > 0) {
            uint256 amount = releasable.mul(sharesWithdrawn).div(
                initialTotalSupply
            );
            uint256 _fee = amount.mul(poolFee).div(10000);

            ..SNIP..

            uint256 amountWithdrawn = amount.sub(_fee);
            token.transfer(recipient, amountWithdrawn);

            _erc20Released[address(token)] += amount;
            ..SNIP..
        }
    }
```

Test Scripts

Following is the test output that demonstrates the above scenario:

```
      Contract: Rubicon Exchange and Pools Original Tests
        Deployment
          ✓ is deployed (1783ms)
        Bath House Initialization of Bath Pair and Bath Tokens
          ✓ Bath House is deployed and initialized (66ms)
            new bathWETH! 0x237eda6f0102c1684caEbA3Ebd89e26a79258C6f
          ✓ WETH Bath Token for WETH asset is deployed and initializ
          ✓ Init BathBuddy Vesting Wallet and Add BathBuddy to WETH
          ✓ Bath Pair is deployed and initialized w/ BathHouse (59ms
            undefined
          ✓ Alice deposit 50 WETH to WETH bathTokens (137ms)
            undefined
          ✓ Bob deposit 50 WETH to WETH bathTokens (174ms)
    bathAssetInstance.bonusTokens.length = 1
    bathBuddyInstance (Vesting Wallet) has 1000 DAI
    bathBuddyInstance.vestedAmount(DAI) = 0.000413359788359788
    bathBuddyInstance.vestedAmount(DAI) = 500.000413359788359788 (En
    bathBuddyInstance.vestedAmount(DAI) = 998.512318121693121693 (La
    0 DAI has been released from BathBuddy Vesting Wallet
    Charles has 8500 bathWETH token, 0 DAI, 0 WETH
    Charles withdraw all his bathWETH tokens
    997.338978147402060445 DAI has been released from BathBuddy Vest
    Charles has 0 bathWETH token, 997.039776453957839827 DAI, 8497.4
    Alice has 5 bathWETH token, 0 DAI, 0 WETH
    998.075233164534207763 DAI has been released from BathBuddy Vest
    Alice has 0 bathWETH token, 0.736034140627007674 DAI, 6.2731175 
          ✓ Add Rewards (100 DAI) to BathBuddy Vesting Wallet  (749m
    bathAssetInstance: underlyingBalance() = 6.2768825 WETH, balance
          ✓ [Debug]
```

Attacker Charles deposited 8500 WETH to the pool and withdraw them immediately at the final hour, and obtained almost all of the `bonusTokens` (997 DAI). When Alice withdraw from the pool, she only received 0.7 DAI as `bonusTokens` .

Script can be found
https://gist.github.com/xiaoming9090/2252f6b6f7e62fca20ecfbaac6f754f5

Note: Due to some unknown issue with the testing environment, please create a new `BathBuddy.released2` functions to fetch the amount of token already released.

Impact

Loss of Fund for the users. BathToken LPs not able to receive the accured `bonusToken` that they are entitled to.

Recommended Mitigation Steps

Update the reward mechanism to ensure that the `bonusTokens` are distribute fairly and rewards of each user are accured correctly.

In the above example, since Alice holds 50% of the shares in the pool throughout the majority of the reward period, she should be entitled to close to 50% to the rewards/bonus. Anyone who joins the pool at the last hour of the reward period should only be entitled to dust amount of `bonusToken`.

Additionally, "spot" (or current) share of the pool should not be used to determine the amount of `bonusToken` a user is entitled to as it is vulnerable to pool/share manipulation or flash-loan attack. Checkpointing mechanism should be implemented so that at the minimum, the user's amount of share in the previous block is used for determining the rewards. This make flash-loan attack infeasible as such attack has to happen within the same block/transaction.

For distributing bonus/rewards, I would suggest checking out a widely referenced Synthetix's Reward Contract as I think that it would be more relevant than OZ's Vesting Wallet for this particular purpose.

bghughes (Rubicon) confirmed

HickupHH3 (judge) commented:

> Great writeup and POC from the warden! **#71** is a little similar, but instead of a flash loan, uses a different method of repeated deposits and withdrawals to achieve the same result.

> Because of the higher quality of this report, I'm using it as the primary issue.

[H-04] First depositor can break minting of shares

*Submitted by MiloTruck, also found by cccz, oyc*109, VAD37, PP1004, SmartSek, minhquanym, unforgiven, berndartmueller, WatchPug, CertoraInc, and sorrynotsorry_

The attack vector and impact is the same as TOB-YEARN-003, where users may not receive shares in exchange for their deposits if the total asset amount has been manipulated through a large "donation".

Proof of Concept

In `BathToken.sol:569-571`, the allocation of shares is calculated as follows:

```
(totalSupply == 0) ? shares = assets : shares = (
    assets.mul(totalSupply)
).div(_pool);
```

An early attacker can exploit this by:

- Attacker calls `openBathTokenSpawnAndSignal()` with `initialLiquidityNew = 1`, creating a new bath token with `totalSupply = 1`
- Attacker transfers a large amount of underlying tokens to the bath token contract, such as `1000000`
- Using `deposit()`, a victim deposits an amount less than `1000000`, such as `1000`:
    - `assets = 1000`
    - `(assets * totalSupply) / _pool = (1000 * 1) / 1000000 = 0.001`, which would round down to `0`
    - Thus, the victim receives no shares in return for his deposit

To avoid minting 0 shares, subsequent depositors have to deposit equal to or more than the amount transferred by the attacker. Otherwise, their deposits accrue to the attacker who holds the only share.

```
it("Victim receives 0 shares", async () => {
    // 1. Attacker deposits 1 testCoin first when creating the l
    const initialLiquidityNew = 1;
    const initialLiquidityExistingBathToken = ethers.utils.parse

    // Approve DAI and testCoin for bathHouseInstance
    await testCoin.approve(bathHouseInstance.address, initialLiq
        from: attacker,
    });
    await DAIInstance.approve(
        bathHouseInstance.address,
        initialLiquidityExistingBathToken,
        { from: attacker }
    );

    // Call open creation function, attacker deposits only 1 tes
```

```
        const desiredPairedAsset = await DAIInstance.address;
        await bathHouseInstance.openBathTokenSpawnAndSignal(
            await testCoin.address,
            initialLiquidityNew,
            desiredPairedAsset,
            initialLiquidityExistingBathToken,
            { from: attacker }
        );

        // Retrieve resulting bathToken address
        const newbathTokenAddress = await bathHouseInstance.getBathT
        const _newBathToken = await BathToken.at(newbathTokenAddress

        // 2. Attacker deposits large amount of testCoin into liquid
        let attackerAmt = ethers.utils.parseUnits("1000000", decimal
        await testCoin.approve(newbathTokenAddress, attackerAmt, {fr
        await testCoin.transfer(newbathTokenAddress, attackerAmt, {f

        // 3. Victim deposits a smaller amount of testCoin, receives
        // In this case, we use (1 million - 1) testCoin
        let victimAmt = ethers.utils.parseUnits("999999", decimals);
        await testCoin.approve(newbathTokenAddress, victimAmt, {from
        await _newBathToken.deposit(victimAmt, victim, {from: victim

        assert.equal(await _newBathToken.balanceOf(victim), 0);
    });
```

Recommended Mitigation Steps

- Uniswap V2 solved this problem by sending the first 1000 LP tokens to the zero address. The same can be done in this case i.e. when `totalSupply() == 0`, send the first min liquidity LP tokens to the zero address to enable share dilution.
- In `_deposit()`, ensure the number of shares to be minted is non-zero:

```
require(shares != 0, "No shares minted");
```

bghughes (Rubicon) confirmed and commented:

> Great issue, what do y'all think of this code snippet as a solution:

> `/// @notice Deposit assets for the user and mint Bath Token shares to receiver function _deposit(uint256 assets, address receiver) internal returns (uint256 shares) { uint256 _pool = underlyingBalance(); uint256 _before = underlyingToken.balanceOf(address(this));

```
        // **Assume caller is depositor**
        underlyingToken.safeTransferFrom(msg.sender, address(this), 
        uint256 _after = underlyingToken.balanceOf(address(this));
        assets = _after.sub(_before); // Additional check for deflat

        if (totalSupply == 0) {
            uint minLiquidityShare = 10**3;
            shares = assets.sub(minLiquidityShare);
            // Handle protecting from an initial supply spoof attack
            _mint(address(0), (minLiquidityShare));
        } else {
            shares = (assets.mul(totalSupply)).div(_pool);
        }

        // Send shares to designated target
        _mint(receiver, shares);

        require(shares != 0, "No shares minted");
        emit LogDeposit(
            assets,
            underlyingToken,
            shares,
            msg.sender,
            underlyingBalance(),
            outstandingAmount,
            totalSupply
        );
        emit Deposit(msg.sender, msg.sender, assets, shares);
    }
```

> `

HickupHH3 (judge) commented:

> LGTM :P

[H-05] BathToken LPs Unable To Receive Bonus Token Due To Lack Of Wallet Setter Method

*Submitted by xiaoming90, also found by 0xNoah, PP1004, sseefried, reassor, hubble, pauliax, sashike*th, and shenwilly_

BathBuddy is a Vesting Wallet that payout withdrawers any `bonusTokens` they may have accrued while staking in the Bath Token (e.g. network incentives/governance tokens).

BathBuddy Vesting Wallet releases a user their relative share of the pool's total vested bonus token during the withdraw call on BathToken.sol. This vesting occurs linearly over Unix time.

It was observed that the BathToken LPs are unable to receive any bonus tokens from the BathBuddy Vesting Wallet during withdraw and the bonus tokens are struck in the BathBuddy Vesting Wallet.

Proof of Concept

The following shows that the address of the BathBuddy Vesting Wallet is stored in the `rewardsVestingWallet` state variable and it is used to call the `release` function to distribute bonus to the BathToken withdrawers.

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L629

```
    function distributeBonusTokenRewards(
        address receiver,
        uint256 sharesWithdrawn,
        uint256 initialTotalSupply
    ) internal {
        if (bonusTokens.length > 0) {
            for (uint256 index = 0; index < bonusTokens.length; inde
                IERC20 token = IERC20(bonusTokens[index]);
                // Note: Shares already burned in Bath Token _withdr

                // Pair each bonus token with a lightly adapted OZ V
                //  are released their relative share of this pool, 
                // The BathBuddy pool should accrue ERC-20 rewards j
                if (rewardsVestingWallet != IBathBuddy(0)) {
                    rewardsVestingWallet.release(
                        (token),
                        receiver,
                        sharesWithdrawn,
                        initialTotalSupply,
                        feeBPS
                    );
                }
            }
        }
    }
```

However, there is no setter method to initialise the value of the `rewardsVestingWallet` state variable in the contracts. Therefore, the value of `rewardsVestingWallet` will always be zero. Note that Solidity only create a default getter for public state variable, but does not create a default setter.

Since `rewardsVestingWallet` is always zero, the condition `if (rewardsVestingWallet != IBathBuddy(0))` will always be evaluated as `false`. Thus, the code block `rewardsVestingWallet.release` will never be reached.

Impact

Loss of Fund for the users. BathToken LPs are not able to receive their `bonusToken`.

Recommended Mitigation Steps

Implement a setter method for the `rewardsVestingWallet` state variable in the contracts so that it can be initialised with BathBuddy Vesting Wallet address.

bghughes (Rubicon) confirmed

[H-06] RubiconRouter _swap does not pass whole amount to RubiconMarket

Submitted by kenzo, also found by IIIIIII, PP1004, blackscale, and hansfriese

When swapping amongst multiple pairs in RubiconRouter's `_swap`, the fee is wrongly accounted for.

Impact

Not all of the user's funds would be forwarded to RubiconMarket, therefore the user would lose funds.

Proof of Concept

The `_swap` function is calculating the pay amount to send to RubiconMarket.sellAllAmount [to be](#):

```
currentAmount.sub(currentAmount.mul(expectedMarketFeeBPS).div(10
```

But this would lead to not all of the funds being pulled by RubiconMarket. I mathematically show this in [this image](#). The correct parameter that needs to be sent to sellAllAmount is:

```
currentAmount.sub(currentAmount.mul(expectedMarketFeeBPS).div(10
```

I mathematically prove this in [this image](#).

Recommended Mitigation Steps

Change the parameter to the abovementioned one.

[bghughes (Rubicon) confirmed](#)

[HickupHH3 (judge) commented](#):

> For the benefit of readers who aren't as math savvy, let's work this out with a numerical example.

> Let's assume a 1% fee: `expectedMarketFeeBPS = 100`. The RubiconMarket charges and pulls this fee separately, so if I have a trade amount of 100, what would be the actual amount to pass into the function?

> The current implementation is `100 - 1% * 100 = 100 - 1 = 99`. However, if that's the case, the market charges 1% of 99 instead, which is 0.99. Hence, the total amount used is `99 + 0.99 = 99.99`, leaving a dust amount of `0.01`.

> Thus, as the warden has proven mathematically, the formula should be `100 - 100 * 100 / (10_000 + 100) ~= 99.0099`. Then, the 1% fee charged is `0.990099...`, making the total approximately equal to 100 (rounding errors).

[H-07] RubiconRouter.swapEntireBalance() doesn't handle the slippage check properly

Submitted by Ruhum, also found by IllIllI, berndartmueller, eccentricexit, blackscale, and hansfriese

The `swapEntireBalance()` function allows the user to pass a `buy_amt_min` value which is the minimum number of tokens they should receive from the swap. But, the function doesn't pass the value to the underlying `swap()` function. Thus, the user's min value will be ignored. Since that will result in unexpected outcomes where user funds might be lost, I rate this issue as HIGH.

Proof of Concept

swapEntireBalance():

```
    function swapEntireBalance(
        uint256 buy_amt_min,
        address[] calldata route, // First address is what is be
        uint256 expectedMarketFeeBPS
    ) external returns (uint256) {
        //swaps msg.sender entire balance in the trade
        uint256 maxAmount = ERC20(route[0]).balanceOf(msg.sender
        ERC20(route[0]).transferFrom(
            msg.sender,
            address(this),
            maxAmount // Account for expected fee
        );
        return
            _swap(
                maxAmount,
                maxAmount.sub(buy_amt_min.mul(expectedMarketFeeB
                route,
                expectedMarketFeeBPS,
                msg.sender
            );
    }
```

The second parameter of the `_swap()` call should be the min out value. Instead `maxAmount.sub(buy_amt_min.mul(expectedMarketFeeBPS).div(10000))` is used.

Example:

```
amount = 100
buy_amt_min = 99
expectedMarketFeeBPS = 500 // 5%

actual buy_amy_min = 100 - (99 * (500 / 10000)) = 95.05
```

So instead of using `99` the function uses `95.05` which could result in the user receiving fewer tokens than they expected.

Recommended Mitigation Steps

Pass `buy_amt_min` directly to `_swap()`.

bghughes (Rubicon) marked as duplicate:

> Duplicate of **#104**.

HickupHH3 (judge) commented:

> Not a duplicate. This has to do with applying a fee on `buy_amt_min` instead of passing the actual value directly. Lower slippage tolerance means potential loss of funds, hence the high severity.

[H-08] Ineffective ReserveRatio Enforcement

Submitted by xiaoming90, also found by shenwilly and pedroais

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L324

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L160

Background

Per whitepaper, ReserveRatio ensures that some amount of pool liquidity is present in the contract at all times. This protects the pools from overutilization by strategists and ensures that a portion of the underlying pool assets are liquid so LPs can withdraw. If the ReserveRatio is set to 50, meaning 50% of a liquidity pool's assets must remain in the pool at all times.

However, it was possible for the strategists to bypass the Reserve Ratio restriction and utilize all the funds in the pools, causing the pools to be illiquid.

Proof of Concept

Strategists place their market making trades via the `BathPair.placeMarketMakingTrades` function. This function would first check if the pool's reserveRatio is maintained before proceeding. If true, strategists will be allowed to place their market making trades with orders with arbitrary pay amount. Strategists could place ask and bid orders with large pay amount causing large amount of funds to be withdrawn from the pools. The root cause is that at the end of the transaction, there is no additional validation to ensure the pools are not overutilized by strategists and the reserve ratio of the pools is maintained.

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L324

```
function placeMarketMakingTrades(
    address[2] memory tokenPair, // ASSET, Then Quote
    uint256 askNumerator, // Quote / Asset
    uint256 askDenominator, // Asset / Quote
    uint256 bidNumerator, // size in ASSET
    uint256 bidDenominator // size in QUOTES
) public onlyApprovedStrategist(msg.sender) returns (uint256 id)
    // Require at least one order is non-zero
    require(
        (askNumerator > 0 && askDenominator > 0) ||
            (bidNumerator > 0 && bidDenominator > 0),
        "one order must be non-zero"
    );

    address _underlyingAsset = tokenPair[0];
    address _underlyingQuote = tokenPair[1];

    (
        address bathAssetAddress,
        address bathQuoteAddress
    ) = enforceReserveRatio(_underlyingAsset, _underlyingQuote);

    require(
        bathAssetAddress != address(0) && bathQuoteAddress != ad
        "tokenToBathToken error"
    );
    .. SNIP..
    // Place new bid and/or ask
    // Note: placeOffer returns a zero if an incomplete order
    uint256 newAskID = IBathToken(bathAssetAddress).placeOffer(
        ask.pay_amt,
        ask.pay_gem,
        ask.buy_amt,
        ask.buy_gem
    );

    uint256 newBidID = IBathToken(bathQuoteAddress).placeOffer(
        bid.pay_amt,
        bid.pay_gem,
        bid.buy_amt,
        bid.buy_gem
    );
    .. SNIP..
}
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L160

```
    /// @notice This function enforces that the Bath House reserveRa
    /// @dev This function should ensure that reserveRatio % of the
    function enforceReserveRatio(
        address underlyingAsset,
        address underlyingQuote
    )
        internal
        view
        returns (address bathAssetAddress, address bathQuoteAddress)
    {
        bathAssetAddress = IBathHouse(bathHouse).tokenToBathToken(
            underlyingAsset
        );
        bathQuoteAddress = IBathHouse(bathHouse).tokenToBathToken(
            underlyingQuote
        );
        require(
            (
                IBathToken(bathAssetAddress).underlyingBalance().mul
                    IBathHouse(bathHouse).reserveRatio()
                )
            ).div(100) <= IERC20(underlyingAsset).balanceOf(bathAsse
            "Failed to meet asset pool reserve ratio"
        );
        require(
            (
                IBathToken(bathQuoteAddress).underlyingBalance().mul
                    IBathHouse(bathHouse).reserveRatio()
                )
            ).div(100) <= IERC20(underlyingQuote).balanceOf(bathQuot
            "Failed to meet quote pool reserve ratio"
        );
    }
```

Test Cases

The following is the snippet of the test case result. Reserve Ratio is initialized to 80% in this example, which means only 20% of the funds could be utilized by strategists. The BathWETH and BathDAI pools contained 1 WETH and 100 DAI respectively after users deposited their funds into the pools. At the bottom half of the output, it shows that it was possible for the strategists to utilise 90% of the funds in the pools to place an ask and bid order, which exceeded the 20% limit.

The last two lines of the output show that 90% of the funds in the pools are outstanding.

```
..SNIP..
---------------- Order Book ----------------
[-] asks index 0: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 1: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 2: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 3: ask_pay_amt = 0, ask_buy_amt = 0
[+] bids index 0: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 1: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 2: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 3: bid_pay_amt = 0, bid_buy_amt = 0
bathAssetInstance: underlyingBalance() = 1 WETH, balanceOf = 1 W
bathQuoteInstance: underlyingBalance() = 100 DAI, balanceOf = 10
After Placing Order
---------------- Order Book ----------------
[-] asks index 0: ask_pay_amt = 0.9, ask_buy_amt = 180
[-] asks index 1: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 2: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 3: ask_pay_amt = 0, ask_buy_amt = 0
[+] bids index 0: bid_pay_amt = 90, bid_buy_amt = 0.9
[+] bids index 1: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 2: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 3: bid_pay_amt = 0, bid_buy_amt = 0
bathAssetInstance: underlyingBalance() = 1 WETH, balanceOf = 0.1
bathQuoteInstance: underlyingBalance() = 100 DAI, balanceOf = 10
..SNIP..
```

Test Script can be found at https://gist.github.com/xiaoming9090/c4fcd4e967bd7d6940429e5d8e39004d

Impact

Following are the impacts of this issue:

1. Underlying pool assets are overutilized by strategists, causing the pools to be illiquid. Users might not be able to withdraw their funds from the pools as the pools might not have sufficient underlying assets remained as their assets have been deployed to the Rubicon Market.
2. Reserve Ratio is one of the key security parameters to safeguard LP's funds so that the amount of losses the pools could potentially incur is limited. Without effective reserve ratio enforcement, strategists could deploy ("invest") all the user capital on the Rubicon Market. If the strategist makes a loss from all their orders, the LP would incur significant loss.

Recommended Mitigation Steps

Check that the reserveRatio for each of the underlying liquidity pools (asset and quote bathTokens) is observed before and after function execution.

```
function placeMarketMakingTrades(
    address[2] memory tokenPair, // ASSET, Then Quote
    uint256 askNumerator, // Quote / Asset
    uint256 askDenominator, // Asset / Quote
    uint256 bidNumerator, // size in ASSET
    uint256 bidDenominator // size in QUOTES
) public onlyApprovedStrategist(msg.sender) returns (uint256 id)
    // Require at least one order is non-zero
    require(
        (askNumerator > 0 && askDenominator > 0) ||
            (bidNumerator > 0 && bidDenominator > 0),
        "one order must be non-zero"
    );

    address _underlyingAsset = tokenPair[0];
    address _underlyingQuote = tokenPair[1];

    (
        address bathAssetAddress,
        address bathQuoteAddress
    ) = enforceReserveRatio(_underlyingAsset, _underlyingQuote);

    require(
        bathAssetAddress != address(0) && bathQuoteAddress != ad
        "tokenToBathToken error"
    );
    .. SNIP..
    // Place new bid and/or ask
    // Note: placeOffer returns a zero if an incomplete order
    uint256 newAskID = IBathToken(bathAssetAddress).placeOffer(
        ask.pay_amt,
        ask.pay_gem,
        ask.buy_amt,
        ask.buy_gem
    );

    uint256 newBidID = IBathToken(bathQuoteAddress).placeOffer(
        bid.pay_amt,
        bid.pay_gem,
        bid.buy_amt,
        bid.buy_gem
    );
    .. SNIP..

    // Ensure that the strategist does not overutilize
    enforceReserveRatio(_underlyingAsset, _underlyingQuote);
}
```

bghughes (Rubicon) confirmed and commented:

> Good issue! I believe it needs to just be moved to the end of the function. Nice catch and already implemented in practice.

[H-09] `BathPair.sol#rebalancePair()` can be front run to steal the pending rebalancing amount

Submitted by WatchPug

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L756-L759

```
    function underlyingBalance() public view returns (uint256) {
        uint256 _pool = IERC20(underlyingToken).balanceOf(address(th
        return _pool.add(outstandingAmount);
    }
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L294-L303

```
    function removeFilledTradeAmount(uint256 amt) external onlyPair
        outstandingAmount = outstandingAmount.sub(amt);
        emit LogRemoveFilledTradeAmount(
            IERC20(underlyingToken),
            amt,
            underlyingBalance(),
            outstandingAmount,
            totalSupply
        );
    }
```

For `BathToken` , there will be non-underlyingToken assets sitting on the contract that have filled to the contract and are awaiting rebalancing by strategists.

We assume the rebalance will happen periodically, between one rebalance to the next rebalance, `underlyingBalance()` will decrease over time as the orders get filled, so that the price per share will get lower while the actual equity remain relatively stable. This kind of price deviation will later be corrected by rebalancing.

Every time a `BathPair.sol#rebalancePair()` get called, there will be a surge of price per share for the `BathToken` , as a certain amount of `underlyingToken` will be transferred into the contract.

This enables a well known attack vector, which allows the pending yields to be stolen by front run the strategist's `BathPair.sol#rebalancePair()` transaction, deposit and take a large share of the vault, and `withdraw()` right after the `rebalancePair()` transaction for instant profit.

Proof of Concept

Given:

- Current `underlyingBalance()` is `100,000 USDC` ;
- Pending rebalancing amount is `1000 USDC` ;
- `strategist` calls `rebalancePair()` ;
- The attacker sends a deposit tx with a higher gas price to deposit `100,000 USDC` , take 50% share of the pool;
- After the transaction in step 1 is mined, the attacker calls `withdraw()` and retireve `100,500 USDC` .

As a result, the attacker has stolen half of the pending yields in about 1 block of time.

Recommendation

Consider adding a new variable to track rebalancingAmount on `BathToken`.

`BathToken` should be notified for any pending rebalancing amount changes via `BathPair` in order to avoid sudden surge of pricePerShare over `rebalancePair()`.

`rebalancingAmount` should be considered as part of `underlyingBalance()`.

bghughes (Rubicon) disputed and marked as duplicate:

> Bad issue due to #344 #43 #74

HickupHH3 (judge) commented:

> It's kinda like the flip side to #341, where an incoming deposit benefits by frontrunning.

> #221 briefly mentions it: "Similar problem also affect the deposit function since it relies on the proper accounting of the underlying balance or outstanding amount too. The amount of BathToken (e.g. BathWETH) that depositer received might affected."

> In this case, a depositor can execute the frontrun attack vector exists **even if the strategist is actively rebalancing.** Hence, the high severity rating is justified.

[H-10] `BathToken.sol#_deposit()` attacker can mint more shares with re-entrancy from hookable tokens

Submitted by WatchPug

`BathToken.sol#_deposit()` calculates the actual transferred amount by comparing the before and after balance, however, since there is no reentrancy guard on this function, there is a risk of re-entrancy attack to mint more shares.

Some token standards, such as ERC777, allow a callback to the source of the funds (the `from` address) before the balances are updated in `transferFrom()`. This callback could be used to re-enter the function and inflate the amount.

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L557-L568

```
    function _deposit(uint256 assets, address receiver)
        internal
        returns (uint256 shares)
    {
        uint256 _pool = underlyingBalance();
        uint256 _before = underlyingToken.balanceOf(address(this));

        // **Assume caller is depositor**
        underlyingToken.transferFrom(msg.sender, address(this), asse
        uint256 _after = underlyingToken.balanceOf(address(this));
        assets = _after.sub(_before); // Additional check for deflat
        ...
```

Proof of Concept

With a ERC777 token by using the ERC777TokensSender `tokensToSend` hook to re-enter the `deposit()` function.

Given:

- `underlyingBalance()`: `100_000e18 XYZ`.
- `totalSupply`: `1e18`

The attacker can create a contract with `tokensToSend()` function, then:

1. `deposit(1)`

```
    -   preBalance  = `100_000e18`;
    -   `underlyingToken.transferFrom(msg.sender, address(this), 1)`
```

2. reenter using `tokensToSend` hook for the 2nd call: `deposit(1_000e18)`

   - preBalance = `100_000e18`;

    - `underlyingToken.transferFrom(msg.sender, address(this), 1_000e18)`
    - postBalance = `101_000e18`;
    - assets (actualDepositAmount) = `101_000e18 - 100_000e18 = 1_000e18`;
    - mint `1000` shares;
3. continue with the first `deposit()` call:
    - `underlyingToken.transferFrom(msg.sender, address(this), 1)`
    - postBalance = `101_000e18 + 1`;
    - assets (actualDepositAmount) = `(101_000e18 + 1) - 100_000e18 = 1_000e18 + 1`;
    - mint `1000` shares;

As a result, with only `1 + 1_000e18` transferred to the contract, the attacker minted `2_000e18 XYZ` worth of shares.

Recommendation

Consider adding `nonReentrant` modifier from OZ's `ReentrancyGuard`.

[bghughes (Rubicon) marked as duplicate and commented]:

> Duplicate of [#283] [#410]. Note that no ERC777 tokens will be created and this will be patched, making it a non-issue in practice.

[HickupHH3 (judge) commented]:

> Not sure what is meant by "no ERC777 tokens will be created", since it's transferring the underlying token which is an arbitrary ERC20, and by extension, ERC777.

> The best practice is to break the CEI pattern for deposits and perform the interaction first. Or simply add reentrancy guards.

[bghughes (Rubicon) confirmed]

Medium Risk Findings (34)

[M-01] Use `call()` instead of `transfer()` when transferring ETH in RubiconRouter

Submitted by Ruhum, also found by cccz, kenzo, xiaoming90, SmartSek, 0x1f8b, IllIllI, GimelSec, kenta, joestakey, berndartmueller, JMukesh, PP1004, Dravee, gzeon, shenwilly, sorrynotsorry, and z3s

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconRouter.sol#L356

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconRouter.sol#L374

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconRouter.sol#L434

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconRouter.sol#L451

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconRouter.sol#L491

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconRouter.sol#L548

Impact

When transferring ETH, use `call()` instead of `transfer()`.

The `transfer()` function only allows the recipient to use 2300 gas. If the recipient uses more than that, transfers will fail. In the future gas costs might change increasing the likelihood of that happening.

Keep in mind that `call()` introduces the risk of reentrancy. But, as long as the router follows the checks effects interactions pattern it should be fine. It's not supposed to hold any tokens anyway.

Proof of Concept

See the linked code snippets above.

Recommended Mitigation Steps

Replace `transfer()` calls with `call()`. Keep in mind to check whether the call was successful by validating the return value:

```
(bool success, ) = msg.sender.call{value: amount}("");
require(success, "Transfer failed.")
```

bghughes (Rubicon) confirmed

[M-02] `withdrawForETH` could be used to drain the WETH in `RubiconRouter.sol`

Submitted by dipp, also found by 0x1f8b, csanuragjain, and pedroais

In the `withdrawForETH` function in `RubiconRouter.sol`, the `targetPool` may be any contract that implements the `IBathToken` interface and returns `wethAddress` as its underlying token. The `withdrawnWETH` amount could be set to the `RubiconRouter.sol` contract's WETH balance so that the contract's entire WETH balance is withdrawn, as long as the `tagetPool` does not transfer any WETH to `RubiconRouter.sol`. The caller of the `withdrawForETH` function would then receive the withdraw amount.

Proof of Concept

```
    function withdrawForETH(uint256 shares, address targetPool)
        external
        payable
        returns (uint256 withdrawnWETH)
    {
        IERC20 target = IBathToken(targetPool).underlyingToken()
        require(target == ERC20(wethAddress), "target pool not w
        require(
            IBathToken(targetPool).balanceOf(msg.sender) >= shar
            "don't own enough shares"
        );
        IBathToken(targetPool).transferFrom(msg.sender, address(
        withdrawnWETH = IBathToken(targetPool).withdraw(shares);
        WETH9(wethAddress).withdraw(withdrawnWETH);

        //Send back withdrawn native eth to sender
        msg.sender.transfer(withdrawnWETH);
    }
```

1. Let `shares` be equal to the contracts WETH balance.
2. The malicious `targetPool` contract returns the `wethAddress` as the underlying token on line 480.
3. `targetPool` returns the max uint256 value for its balanceOf function to pass the require condition on line 483 for any value of shares.
4. The transferFrom on line 486 does not have to do anything and its withdraw function should return the WETH balance of `RubiconRouter.sol`.
5. The `RubiconRouter.sol` contract will then withdraw ETH equal to the `withdrawWETH` amount, which should be equal to the contract's WETH balance.
6. The caller of the `withdrawForETH` function receives the withdraw ETH without providing any WETH.

Recommended Mitigation Steps

Check the contract's WETH balance before the caller is supposed to send the WETH and after the WETH is sent to confirm the contract has received enough WETH from the caller.

bghughes (Rubicon) confirmed

HickupHH3 (judge) commented:

> Will keep this as medium severity because of the pre-requisite of users accidentally sending ETH to the router contract.

[M-03] Use `safeTransfer()` / `safeTransferFrom()` instead of `transfer()` / `transferFrom()`

Submitted by berndartmueller, also found by cccz, oyc109, aez121, Ruhum, MaratCerby, fatherOfBlocks, antonttc, VAD37, kenzo, xiaoming90, jayjonah8, SmartSek, PP1004, cryptphi, shenwilly, gzeon, IllIllI, ACai, GimelSec, kenta, blockdev, JMukesh, Bahurum, joestakey, throttle, WatchPug, ilan, 0xDjango, CertoraInc, peritoflores, 0xsomeone, horsefacts, Deivitto, camden, pedroais, broccolirob, _Adam, 0x1f8b, BouSalman, defsec, dipp, Dravee, ellahi, Kaiziron, minhquanym, pauliax, sashik_eth, simon135, and z3s

It is a good idea to add a `require()` statement that checks the return value of ERC20 token transfers or to use something like OpenZeppelin's `safeTransfer()` / `safeTransferFrom()` unless one is sure the given token reverts in case of a failure. Failure to do so will cause silent failures of transfers and affect token accounting in contract.

However, using `require()` to check transfer return values could lead to issues with non-compliant ERC20 tokens which do not return a boolean value. Therefore, it's highly advised to use OpenZeppelin's `safeTransfer()` / `safeTransferFrom()`.

Proof of Concept

RubiconRouter.sol

L251: `ERC20(route[route.length - 1]).transfer(to, currentAmount);`
L303: `ERC20(buy_gem).transfer(msg.sender, fill);`
L320: `ERC20(buy_gem).transfer(msg.sender, fill);`
L348: `ERC20(buy_gem).transfer(msg.sender, buy_amt);`
L377: `ERC20(pay_gem).transfer(msg.sender, max_fill_amount - fill);`
L406: `ERC20(buy_gem).transfer(msg.sender, _after - _before);`
L471: `ERC20(targetPool).transfer(msg.sender, newShares);`

peripheral_contracts/BathBuddy.sol

L114: `token.transfer(recipient, amountWithdrawn);`

rubiconPools/BathPair.sol

L601: `IERC20(asset).transfer(msg.sender, booty);`

L615: `IERC20(quote).transfer(msg.sender, booty);`

rubiconPools/BathToken.sol

L353: `IERC20(filledAssetToRebalance).transfer(`

L357: `IERC20(filledAssetToRebalance).transfer(msg.sender, stratReward);`

L602: `underlyingToken.transfer(feeTo, _fee);`

L605: `underlyingToken.transfer(receiver, amountWithdrawn);`

Recommended Mitigation Steps

Consider using `safeTransfer()` / `safeTransferFrom()` instead of `transfer()` / `transferFrom()` .

bghughes (Rubicon) confirmed

[M-04] RubiconRouter: Excess ether did not return to the user

Submitted by cccz, also found by fatherOfBlocks, Ruhum, csanuragjain, gzeon, IllIIII, GimelSec, dipp, berndartmueller, pedroais, horsefacts, AlleyCat, Bahurum, and shenwilly

In swapWithETH/buyAllAmountWithETH/offerWithETH/depositWithETH functions of the RubiconRouter contract, when `msg.value > max_fill_withFee/pay_amt/amount/amtWithFee` , the excess ether will not be returned to the user.

Proof of Concept

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L325-L339

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L383-L393 https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L455-L462

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L494-L507

Recommended Mitigation Steps

Return excess ether to msg.sender, or require msg.value == max*fill*withFee/pay_amt/amount/amtWithFee.

bghughes (Rubicon) confirmed

KenzoAgada (warden) commented:

> Marking this as main for all issues with various functions in RubiconRouter not sending excess ether back to the user as this issue contains all the examples.

[M-05] Inconsistent Order Book Accounting When Working With Transfer-On-Fee or Deflationary Tokens

Submitted by xiaoming90, also found by MaratCerby, IllIIl, GimelSec, PP1004, blockdev, berndartmueller, WatchPug, and ilan

A transfer-on-fee token or a deflationary/rebasing token, causing the received amount to be less than the accounted amount. For instance, a deflationary tokens might charge a certain fee for every transfer() or transferFrom().

Rubicon Finance supports the trading of any ERC20 token, and anyone can liquidity pool for a new token. Thus, it is possible that such a transfer-on-fee token or a deflationary/rebasing token be used in the protocol.

Based on the source code and comment of `BathToken._deposit()` , it appears that the team is aware of this issue, and proactively implemented control (before & after balance checks) to deal with deflationary tokens.

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L557

```
    function _deposit(uint256 assets, address receiver)
        internal
        returns (uint256 shares)
    {
        uint256 _pool = underlyingBalance();
        uint256 _before = underlyingToken.balanceOf(address(this));

        // **Assume caller is depositor**
        underlyingToken.transferFrom(msg.sender, address(this), asse
        uint256 _after = underlyingToken.balanceOf(address(this));
        assets = _after.sub(_before); // Additional check for deflat

        (totalSupply == 0) ? shares = assets : shares = (
            assets.mul(totalSupply)
        ).div(_pool);

        // Send shares to designated target
        _mint(receiver, shares);
        ..SNIP..
    }
```

However, such control was not consistently applied across the protocol, and might cause the internal accounting of the orderbook to be incorrect.

Proof of Concept

If the `pay_gem` token is an deflationary token, the `info.pay_amt` and the actual amount of `pay_gem` tokens received will not be in sync.

For instance, assume that XYZ token is a deflation token that charges 10% fee for every transfer. If an `offer(100, XYZ, 100, DAI)` is executed, an order with 100 XYZ (pay) and 100 DAI (buy) will be added to the orderbook. However, the orderbook will only received 90 XYZ, thus only 90 XYZ is ecrowed in the orderbook. This discrepancy would break the internal accounting system of the order book.

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMa rket.sol#L392

```
    /// @notice Key function to make a new offer. Takes funds from tl
    function offer(
        uint256 pay_amt,
        ERC20 pay_gem,
        uint256 buy_amt,
        ERC20 buy_gem
    ) public virtual can_offer synchronized returns (uint256 id) {
        ..SNIP..
        OfferInfo memory info;
        info.pay_amt = pay_amt;
        info.pay_gem = pay_gem;
        info.buy_amt = buy_amt;
        info.buy_gem = buy_gem;
        info.owner = msg.sender;
        info.timestamp = uint64(block.timestamp);
        id = _next_id();
        offers[id] = info;

        require(pay_gem.transferFrom(msg.sender, address(this), pay_
        ..SNIP..
    }
```

Impact

The internal accounting system of the order book would be inaccurate or break, affecting the protocol operation.

Recommended Mitigation Steps

In the `offer` function, get the actual received amount by calculating the difference of token balance before and after the transfer, and set the `info.pay_amt` to the actual received amount.

Alternatively, the team might want to consider implementing whitelisting mechanism so that deflationary tokens will not be supported if the risk of allowing permissionless creation of pool with arbitrary token deems to be significant. A DAO may be formed in the future to manage the whitelisting.

bghughes (Rubicon) marked as duplicate and commented:

> Multiple issues regarding the potential for deflationary tokens or Fee-on-transfer as a global ERC-20 issue, not sure how to handle this.

HickupHH3 (judge) commented:

> Let's take this issue as the primary issue, will mark related issues as duplicates.

> Regarding handling of FoT tokens, it typically is more concerning for incoming token transfers (deposits in this case). What DEXes usually do is to explicitly mention that FoT tokens are not supported (as in the case of UniV3) in documentation, or, as suggested, get the actual amount received by calculating the difference of token balance before and after the transfer.

bghughes (Rubicon) confirmed and commented:

> Fantastic thank you. I think we will just not support these for the time being.

[M-06] Cannot deposit to BathToken if token is Deflationary Token (BathHouse.sol)

Submitted by PP1004, also found by unforgiven, GimelSec, and camden

Function `openBathTokenSpawnAndSignal` will alway revert when `newBathTokenUnderlying` or `desiredPairedAsset` is deflationary token

Proof of Concept

There are ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()` For example, I will assume that `newBathTokenUnderlying` is deflationary token. After line 163, the actual amount of `newBathTokenUnderlying` that BathHouse gained will be smaller than `initialLiquidityNew` . It will make the deposit call reverted because there are not enough fund to transfer.

Recommended Mitigation Steps

set `initialLiquidityNew = newBathTokenUnderlying.balanceOf(address(this))` after line 163 and `initialLiquidityExistingBathToken = desiredPairedAsset.balanceOf(address(this))` after line 178

bghughes (Rubicon) acknowledged and commented:

> This is correct, though I believe un needed. If the user wants to create a vault for a deflationary token they need only account for said transfer fee when calculating their `initialLiquidityNew` value.

HickupHH3 (judge) commented:

> Not sure how you can account for transfer fee in `initialLiquidityNew` since it's the same amount used for approval and deposit: `IBathToken(newOne).deposit(initialLiquidityNew, msg.sender);`

> It simply means that deflationary / FoT tokens arent supported at all, which isn't necessarily a bad thing. There isn't a loss of assets, though `function of the protocol or its availability could be impacted`. Keeping it at medium severity, although could've potentially lowered to QA too.

[M-07] No Storage Gap for Upgradeable Contracts

Submitted by 0x1337, also found by broccolirob

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L448-L449

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L525-L535

Impact

For upgradeable contracts, there must be storage gap to “allow developers to freely add new state variables in the future without compromising the storage compatibility with existing deployments”. Otherwise it may be very difficult to write new implementation code. Without storage gap, the variable in child contract might be overwritten by the upgraded base contract if new variables are added to the base contract. This could have unintended and very serious consequences to the child contracts.

Refer to the bottom part of this article: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable

Proof of Concept

As an example, the `ExpiringMarket` contract inherits `SimpleMarket`, and the `SimpleMarket` contract does not contain any storage gap. If in a future upgrade, an additional variable is added to the `SimpleMarket` contract, that new variable will overwrite the storage slot of the `stopped` variable in the `ExpiringMarket` contract, causing unintended consequences.

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L448-L449

Similarly, the `ExpiringMarket` does not contain any storage gap either, and the `RubiconMarket` contract inherits `ExpiringMarket`. If a new variable is added to the `ExpiringMarket` contract in an upgrade, that variable will overwrite the `buyEnabled` variable in `ExpiringMarket` contract.

Recommended Mitigation Steps

Recommend adding appropriate storage gap at the end of upgradeable contracts such as the below. Please reference OpenZeppelin upgradeable contract templates.

```
    uint256[50] private __gap;
```

bghughes (Rubicon) confirmed

HickupHH3 (judge) commented:

> Out of curiosity, is there a reason why the openzeppelin-upgrades package isn't used?

> According to the README, Transparent Upgradeable Proxies is used to make sure "all contracts can be iterated and improved over time." seems like the `initializable` contract should've been inherited and utilized so that you wouldn't have to worry about adding in storage gaps.

bghughes (Rubicon) commented:

> Optimism had a custom compiler that required custom proxies. This led to the package not working otherwise I would have used it.

> It seems like always extending the top-level storage contract as a practice should avoid any issues here, right? It is a good issue that highlights I should never try to extend inherited contract's storage.

HickupHH3 (judge) commented:

> Ahhh I see. It depends on future upgrades that will affect the storage layout. Inheriting from Ownable and Pausable for instance shouldnt affect upgradeability much because I dont think their required functionality will drastically change.

> But yeah, ensuring there is sufficient gap is to future-proof the contracts. Worst case, do a new deployment.

bghughes (Rubicon) commented:

> I just ran into this attempting to add the Reentrancy Gaurd to a contract. To be clear warden should I add this to the end of the existing base contract? For example, appending the uint256[50] gap to the base contract before inheritance?

> Thanks and good issue.

> I'd love to know the appropriate way to bolt on something like `ReentrancyGuard` onto an existing proxy-wrapped contract - is it even possible?

> My game plan is to bring the `nonReentrant` modifier into the top-level contract to only extend storage. Thank you again warden!

HickupHH3 (judge) commented:

> https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/security/ReentrancyGuardUpgradeable.sol

[M-08] USDT is not supported because of approval mechanism

Submitted by kenzo, also found by MaratCerby, 0x1f8b, berndartmueller, Dravee, 0xsomeone, cryptphi, IllIIl, and xiaoming90

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/rubiconPools/BathHouse.sol#L180

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconRouter.sol#L157

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/rubiconPools/BathToken.sol#L256

Vulnerability details

When using the approval mechanism in USDT, the approval must be set to 0 before it is updated. In Rubicon, when creating a pair, the paired asset's approval is not set to 0 before it is updated.

Impact

Can't create pairs with USDT, the most popular stablecoin, as the approval will revert.

Proof of Concept

USDT reverts on approval if previous allowance is not 0:

```
require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))
```

When creating a pair, Rubicon approves the paired asset without first setting it to 0:

```
    desiredPairedAsset.approve(pairedPool, initialLiquidityExistingBa
```

Therefore, if desiredPairedAsset is USDT, the function will revert, and pairs with USDT can not be created.

This problem will also manifest in RubiconMarket's approval function and BathToken's approval function,

Recommended Mitigation Steps

Set the allowance to 0 before setting it to the new value.

bghughes (Rubicon) disputed and commented:

> This doesn't sound right to me because it implies that USDT is non-compliant with the infinite approval pattern. I believe USDT can be infinitely approved, asking for an extra pair of eyes to verify.

> Just tested this on Rubicon trading and there is no need to re-approve USDT before using it so I believe this is a bad issue.

KenzoAgada (warden) commented:

> @bghughes This is not related to infinite approval (which USDT allows); you're approving a specific amount. USDT on mainnet does not allow changing from a non-zero allowance to another non-zero allowance.

> Did you test it on Optimism? USDT on optimism seems to use an updated version of USDT, unlike the one in mainnet. In Optimism you can change the approval from non-zero to another non-zero. But if you look at USDT on Ethereum mainnet, the code does not allow this (look at the second clause):
>
> ```
>         require(!((_value != 0) && (allowed[msg.sender][_spender
> ```

HickupHH3 (judge) commented:

> The issue is valid on `approveAssetOnMarket()` and `approveMarket()`, but not on the other lines. Even that, it's likely for these functions to be called just once. Nevertheless, I will let the issue stand.

> The reason is because the allowance will be entirely consumed when the deposit function is called, so it will be set to zero always.

```
        desiredPairedAsset.approve(
            pairedPool,
            initialLiquidityExistingBathToken
        );

        // Deposit assets and send Bath Token shares to msg.sender
        // This will use up the entire allowance given

        IBathToken(pairedPool).deposit(
            initialLiquidityExistingBathToken,
            msg.sender
        );
```

> Hence, any issue that does not reference the `approveAssetOnMarket()` and `approveMarket()` functions will be marked as invalid.

bghughes (Rubicon) confirmed and commented:

> Thank you for the explanation and thank you @HickupHH3.

[M-09] BathBuddy locks up Ether it receives

Submitted by Ruhum, also found by 0x1f8b and pauliax

The BathBuddy contract is able to receive ETH. But, there's no way of ever retrieving that ETH from the contract. The funds will be locked up.

Currently, there seems to be no logic in the protocol where ETH is sent to the contract. But, it might happen in the future. So I'd say it's a MED issue.

Proof of Concept

`receive()` function: https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/peripheral_contracts/BathBuddy.sol#L69

Recommended Mitigation Steps

Remove the `receive()` function if the contract isn't supposed to handle ETH. Otherwise, add the necessary logic to release the ETH it gets.

[bghughes (Rubicon) confirmed]()

[M-10] Wrong DOMAIN_SEPARATOR

Submitted by 0x1f8b, also found by rotcivegaf, unforgiven, CertoraInc, eccentricexit, and IllIllI

The `DOMAIN_SEPARATOR` is wrongly calculated.

Proof of Concept

In the `initialize` method of the `BathToken` contract, the `name` of the contract is used to calculate the `DOMAIN_SEPARATOR`, however said name is set later, so it will use an incorrect `name`, making it impossible to calculate the `DOMAIN_SEPARATOR` correctly.

```
    DOMAIN_SEPARATOR = keccak256(
        abi.encode(
            keccak256(
                "EIP712Domain(string name,string version,uint256 cha
            ),
            keccak256(bytes(name)),
            keccak256(bytes("1")),
            chainId,
            address(this)
        )
    );
    name = string(abi.encodePacked(_symbol, (" v1")));
```

Affected source code:

- BathToken.sol#L199-L210

Recommended Mitigation Steps

- Set the `name` before using it.

bghughes (Rubicon) confirmed

[M-11] previewWithdraw calculates shares wrongly

Submitted by kenzo, also found by PP1004, pedroais, and CertoraInc

The fee is wrongly accounted for in `previewWithdraw`.

Impact

Function returns wrong result;

Additionally, `withdraw(assets,to,from)` will always revert. (The user can still withdraw his assets via other functions).

Proof of Concept

The `previewWithdraw` function returns *less* shares than the required assets (notice the substraction):

```
        uint256 amountWithdrawn;
        uint256 _fee = assets.mul(feeBPS).div(10000);
        amountWithdrawn = assets.sub(_fee);
        shares = convertToShares(amountWithdrawn);
```

This won't work, because if the user wants to receive amount of `assets`, he needs to burn *more* shares than that to account for the fee. Not less.

This will also make `withdraw(assets,to,from)` revert, because it takes the amount of shares from `previewWithdraw`, and then checks how much assets were really sent to the user, and verifies that it's at least how much he asked for:

```
            uint256 expectedShares = previewWithdraw(assets);
            uint256 assetsReceived = _withdraw(expectedShares, recei
            require(assetsReceived >= assets, "You cannot withdraw t
```

But since the expectedShares is smaller than the original amount, and since `_withdraw` **deducts** the fee from expectedShares, then always `assets > assetsReceived`, and the function will revert.

Recommended Mitigation Steps

The amount of shares that `previewWithdraw` should return is: `convertToShares(assets.add(assets.mul(feeBPS).div((10000.sub(feeBPS)))))` I prove this mathematically in this image.

bghughes (Rubicon) confirmed

HickupHH3 (judge) commented:

> Keeping it as medium severity because while protocol functionality is impacted, users can withdraw through the `redeem()` function.

bghughes (Rubicon) commented:

> The new solution, thank you, Kenzo.

> `/// @notice * EIP 4626 * function previewWithdraw(uint256 assets) public view returns (uint256 shares) { if (totalSupply == 0) { shares = 0; } else { shares = convertToShares( assets.add(assets.mul(feeBPS).div((uint(10000).sub(feeBPS)))) ); } }`

bghughes (Rubicon) commented:

> Keeping it as medium severity because while protocol functionality is impacted, users can withdraw through the `redeem()` function.

> Note that we use the `withdraw` that relies on msg.sender as the caller in production so were not affected in practice. It's interesting that various wardens

> have different answers to the solution for this issue. This one seems best and I'm going with it for now!

[M-12] Admin rug vectors

*Submitted by IllIllI, also found by oyc*109, 0x1f8b, xiaoming90, SmartSek, 0xDjango, Dravee, pauliax, rotcivegaf, sashik*eth, shenwilly, and StErMi*

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L216-L229

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L334-L337

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L269-L272

Impact

There are multiple functions that the admin can call to brick various parts of the system, leading to user's funds being locked

Even if the owner is benevolent the fact that there is a rug vector available may [negatively impact the protocol's reputation](). See [this]() example where a similar finding has been flagged as a high-severity issue. I've downgraded these instances to be a medium since it requires cooperation of the admin.

Proof of Concept

Here are some examples:

Overwrite state:

```
File: contracts/rubiconPools/BathHouse.sol   #1

216       /// @notice A migration function that allows the admin
217       function adminWriteBathToken(ERC20 overwriteERC20, add
        external
        onlyAdmin
    {
        tokenToBathToken[address(overwriteERC20)] = newBat
        emit LogNewBathToken(
            address(overwriteERC20),
            newBathToken,
            address(0),
            block.timestamp,
            msg.sender
        );
    }
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L216-L229

Steal new funds with new malicious market

```
File: contracts/rubiconPools/BathHouse.sol    #2

    /// @notice Admin-only function to set a Bath Token's
    function setMarket(address newMarket) external onlyAdm
        RubiconMarketAddress = newMarket;
    }
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L334-L337

Add a ton of bonus tokens, making withdrawals break because of unbounded for-loop

```
File: contracts/rubiconPools/BathToken.sol    #3

    /// @notice Admin-only function to add a bonus token t
    function setBonusToken(address newBonusERC20) external
        bonusTokens.push(newBonusERC20);
    }
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L269-L272

Recommended Mitigation Steps

Validate input arguments and require specific upgrade paths for each contract, not just allowing the admin to set whatever they decide, add limit to number of bonus tokens.

bghughes (Rubicon) acknowledged:

> Centralization risk is acknowledged.

[M-13] Early funds withdrawers can get bonus in multiples of vested bonus tokens (e.g. 2-times, 3-times, etc.)

Submitted by hubble, also found by Ruhum

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L270

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L629

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L98-L101

Vulnerability details

The function setBonusToken allows the same BonusToken to be added more than once to the array bonusTokens.

```
    function setBonusToken(address newBonusERC20) external onlyBat
        bonusTokens.push(newBonusERC20);
        }
```

Impact

If that happens, early withdrawers can get Bonus in multiples of what they actually have the right to. Late withdrawers, might not get any Bonus due to shortage.

Proof of Concept

BathToken.sol, function setBonusToken

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L270-L272

1. function setBonusToken allows the same BonusToken to be added more than once to the array.

   BathToken.sol, function

   distributeBonusTokenRewards

   https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L629

2. a. As and when distributeBonusTokenRewards is triggered during a withdraw call, the same bonusToken will be released more than once.

   BathBuddy.sol, function release

   https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L98-L101

3. b. The release function is called.

Recommended Mitigation Steps

Add the required validations to avoid duplicate additions of bonus tokens.

```
    function setBonusToken(address newBonusERC20) external onlyBat
      require(newBonusERC20 != address(0), "invalid_addr");
      if (bonusTokens.length > 0) {
        for (uint256 index = 0; index < bonusTokens.length; index+
          require (token != newBonusERC20, "token already exists")
        }
      }
      bonusTokens.push(newBonusERC20);
    }
```

bghughes (Rubicon) confirmed

HickupHH3 (judge) commented:

> Similar, but different to unbounded tokens case because it’s about duplicates.

[M-14] No cap on fees can result in a DOS in BathToken.withdraw()

*Submitted by Ruhum, also found by cccz, 0x1f8b, kenzo, xiaoming90, reassor, sseefried, catchup, shenwilly, GimelSec, defsec, StErMi, berndartmueller, throttle, sashik*eth, eccentricexit, 0xDjango, peritoflores, pedroais, hubble, joestakey, 0x4non, blackscale, csanuragjain, Dravee, ellahi, horsefacts, hubble, and rotcivegaf_

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconMarket.sol#L1232

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/rubiconPools/BathToken.sol#L261

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/rubiconPools/BathToken.sol#L498-L499

Impact

The owner can set an arbitrary fee. If they set it to a value above 100%, withdrawing BathTokens won’t be possible anymore because of an underflow. In RubiconMarket the owner can also have an arbitrary fee although that won't result in a DOS. But, the user might pay an absurdly high fee.

There should be checks that only allow fees up to a specific value, e.g. 10%.

Proof of Concept

For the DOS:

1. Owner sets fees to 101% using `BathToken.setFeeBPS()`: https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/rubiconPools/BathToken.sol#L261
2. User tries to withdraw which will revert here because `fee > r`: https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/rubiconPools/BathToken.sol#L604

Recommended Mitigation Steps

Add a limit to the constructors where a fee is set and to all the configuration functions for fees.

bghughes (Rubicon) acknowledged

[M-15] Outstanding Amount Of A Pool Reduced Although Tokens Are Not Repaid

Submitted by xiaoming90, also found by shenwilly, unforgiven, and WatchPug

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L213

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L294

Background

In the BathToken Contract, `outstandingAmount` means the amount of tokens that have been taken out of a pool, but have not been repaid back to the pool yet. Additionally, `outstandingAmount` is NOT inclusive of any non-underlyingToken assets sitting on the Bath Tokens that have filled to here and are awaiting rebalancing to the underlyingToken by strategists per code comment.

Placing new trade orders and scrubbing existing trade orders are essential for maintaining an effective market, thus they are the key operation of a Strategist and these activities are expected to be performed frequently.

It was observed that the `outstandingAmount` of the pool was reduced prematurely when a trade order is scrubbed by strategist, thus making the `outstandingAmount` inaccurate.

Walkthrough

The following are the output of test script which will be used to demonstrate the issue.

Assume that Alice deposited 50 WETH and 50 DAI, and Bob deposited 50 WETH and 50 DAI to their respective BathToken pools.

At this point, as shown below, for both BathWETH and BathDAI pool:
underlyingBalance = 100, balanceOf = 100, Outstanding Amount = 0.

These values are correct as the users have deposited 100 and none of the funds have been utilised by the Strategist yet. No asset is escrowed in the orderbook yet. The orderbook is empty.

Note: underlyingBalance = outstandingAmount + balanceOf

```
    --------------- Order Book ---------------
    [-] asks index 0: ask_pay_amt = 0, ask_buy_amt = 0
    [-] asks index 1: ask_pay_amt = 0, ask_buy_amt = 0
    [-] asks index 2: ask_pay_amt = 0, ask_buy_amt = 0
    [-] asks index 3: ask_pay_amt = 0, ask_buy_amt = 0
    [+] bids index 0: bid_pay_amt = 0, bid_buy_amt = 0
    [+] bids index 1: bid_pay_amt = 0, bid_buy_amt = 0
    [+] bids index 2: bid_pay_amt = 0, bid_buy_amt = 0
    [+] bids index 3: bid_pay_amt = 0, bid_buy_amt = 0
    bathAssetInstance: underlyingBalance() = 100 WETH, balanceOf = 1
```

```
bathQuoteInstance: underlyingBalance() = 100 DAI, balanceOf = 10
Order Book Escrowed 0 WETH and 0 DAI
```

The Strategist placed a market trade order, thus an ask and bid order was sent to the orderbook and 40 WETH and 40 DAI were escrowed in the orderbook. Since 40 WETH and 40 DAI were utilised by the Strategist, the pool's `balanceOf` dropped from 100 to 60, and the outstanding amount increased from 0 to 40. The `underlyingBalance` remains at 100.

```
After Placing Order
--------------- Order Book ---------------
[-] asks index 0: ask_pay_amt = 40, ask_buy_amt = 800
[-] asks index 1: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 2: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 3: ask_pay_amt = 0, ask_buy_amt = 0
[+] bids index 0: bid_pay_amt = 40, bid_buy_amt = 4
[+] bids index 1: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 2: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 3: bid_pay_amt = 0, bid_buy_amt = 0
bathAssetInstance: underlyingBalance() = 100 WETH, balanceOf = 6
bathQuoteInstance: underlyingBalance() = 100 DAI, balanceOf = 60
Order Book Escrowed 40 WETH and 40 DAI
```

Charles decided to buy the ask order (OrderID = 1) in the orderbook, thus he paid 800 DAI and received 40 WETH. 40 WETH escrowed in the orderbook was released/transfered to Charles and the 800 DAI that Charles paid was redirected to BathWETH pool waiting to be rebalanced by Strategist. The ask order was removed from the orderbook since it had been fulfilled.

At this point, the outstanding amount of BathWETH was 40 and this was correct since the BathWETH pool was still missing the 40 WETH that was taken out by the Strategist earlier, and had not been repaid back to the pool yet.

Noted that there were 800 DAI sitting on the BathWETH pool, but understood that the `outstandingAmount` is NOT inclusive of any non-underlyingToken assets sitting on the Bath Tokens that have filled to here and are awaiting rebalancing to the underlyingToken by strategists.

```
After Charles brought the best ask order
---------------- Order Book ----------------
[-] asks index 0: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 1: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 2: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 3: ask_pay_amt = 0, ask_buy_amt = 0
[+] bids index 0: bid_pay_amt = 40, bid_buy_amt = 4
[+] bids index 1: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 2: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 3: bid_pay_amt = 0, bid_buy_amt = 0
bathAssetInstance: underlyingBalance() = 100 WETH, balanceOf = 6
bathQuoteInstance: underlyingBalance() = 100 DAI, balanceOf = 60
Order Book Escrowed 0 WETH and 40 DAI
```

Strategist decided to scrub his trade order by calling `BathPair.scrubStrategistTrade()` , so any outstanding orders in the orderbook were cancelled. Therefore, the 40 DAI escrowed in the orderbook was released back to the bathDAI pool. Thus, the bathDAI pool had a `balanceOf` = 100, and no outstanding amount since all the DAI had been repaid back.

However, scrubbing a trade order has a unintended effect on the BathWETH pool. All the outstanding amount in BathWETH pool were cleared (changed from 40 to 0) although no WETH was repaid back to the BathWETH pool. The outstanding amount was inaccurate and did not reflect the actual amount not repaid back to the pool yet.

Per the design, the only time that the WETH will be repaid back to BathWETH pool is when a strategist performs a rebalance to swap (either with other pool or external AMM) the 800 DAI (non-underlying asset) sitting on BathWETH pool.

```
Strategist scrub his trade order
---------------- Order Book ----------------
[-] asks index 0: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 1: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 2: ask_pay_amt = 0, ask_buy_amt = 0
[-] asks index 3: ask_pay_amt = 0, ask_buy_amt = 0
[+] bids index 0: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 1: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 2: bid_pay_amt = 0, bid_buy_amt = 0
[+] bids index 3: bid_pay_amt = 0, bid_buy_amt = 0
bathAssetInstance: underlyingBalance() = 60 WETH, balanceOf = 60
bathQuoteInstance: underlyingBalance() = 100 DAI, balanceOf = 10
```

```
Order Book Escrowed 0 WETH and 0 DAI
```

The inaccurate outstanding amount of a pool casues unintended impact on a number of key functions (e.g. deposit, withdraw) of the protocol as they rely on the proper accounting of the `underlyingBalance` of a pool.

Note: underlyingBalance = outstandingAmount + balanceOf (if outstandingAmount is wrong, the underlyingBalance would be wrong too)

The following shows an example of the impact to Alice due to this issue. Alice who has earlier deposited 50 WETH to the BathWETH pool decided to withdraw all her investment.

Note: Alice holds around 50% of the shares in the bathWETH pool. Fee is charged during pool withdrawal.

- First scenario - If the strategist did not scrub his trade order at the first place, `underlyingBalance` of BathWETH pool will be 100, and she would received around 49 WETH back.
- Second scenario - However, if the strategist srcubs his trade order, the bug will be triggered, and the `underlyingBalance` of BathWETH pool will change from 100 to 60. As shown in the output below, she will only received around 29 WETH if she withdraw all her investment.

```
Alice (accounts[1]) has 50 bathWETH and 50 bathDAI
Alice (accounts[1]) has 0 WETH and 950 DAI
Alice Attempting to withdraw all 50 bathWETH token
Alice Succeed in withdrawing all bathWETH token
Alice (accounts[1]) has 29.991 WETH and 950 DAI
```

Alice lost 20 WETH In the second scenario.

Similar problem also affect the deposit function since it relies on the proper accounting of the underlying balance or outstanding amount too. The amount of BathToken (e.g. BathWETH) that depositer received might be affected.

Proof of Concept

The following aims to explain why this issue occurred.

When `BathPair.scrubStrategistTrade` is called, it will in turn call `BairPair.handleStratOrderAtID`. If an order has been filled, it will call `BathToken.removeFilledTradeAmount`

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L213

```
/// @dev Cancels outstanding orders and manages the ledger of ou
function handleStratOrderAtID(uint256 id) internal {
    StrategistTrade memory info = strategistTrades[id];
    address _asset = info.askAsset;
    address _quote = info.bidAsset;

    address bathAssetAddress = IBathHouse(bathHouse).tokenToBath'
        _asset
    );
    address bathQuoteAddress = IBathHouse(bathHouse).tokenToBath'
        _quote
    );
    order memory offer1 = getOfferInfo(info.askId); //ask
    order memory offer2 = getOfferInfo(info.bidId); //bid
    uint256 askDelta = info.askPayAmt - offer1.pay_amt;
    uint256 bidDelta = info.bidPayAmt - offer2.pay_amt;

    // if real
    if (info.askId != 0) {
        // if delta > 0 - delta is fill => handle any amount of
        if (askDelta > 0) {
            logFill(askDelta, info.strategist, info.askAsset);
            IBathToken(bathAssetAddress).removeFilledTradeAmount
            // not a full fill
            if (askDelta != info.askPayAmt) {
                IBathToken(bathAssetAddress).cancel(
                    info.askId,
                    info.askPayAmt.sub(askDelta)
                );
            }
        }
        // otherwise didn't fill so cancel
        else {
            IBathToken(bathAssetAddress).cancel(info.askId, info
        }
    }
    ..SNIP..
}
```

`BathToken.removeFilledTradeAmount` will reduce the outstanding amount of the pool.

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L294

```
    /// @notice A function called by BathPair to maintain proper acc
    function removeFilledTradeAmount(uint256 amt) external onlyPair
        outstandingAmount = outstandingAmount.sub(amt);
        ..SNIP..
    }
```

Test Script

Following test script is used to generate the output shown in the walkthrough https://gist.github.com/xiaoming9090/f26e0d3ea365edc89b257d6c0aa54ac9

Impact

Loss of fund for the users. Key protocol functions (e.g. deposit and withdraw) that rely on proper accounting of the underlying balance or outstanding amount do not function as expected.

Recommended Mitigation Steps

Update the systems to ensure that the `outstandingAmount` of the pool is reduced only when the outstanding amount of tokens have been repaid back.

For instance, in the above example, the reduction should happen during rebalancing when the 800 DAI sitting on the bathWETH pool have been swapped for WETH, and the swapped WETH are replenished/repaid back to the pool. In this case, the `outstandingAmount` can be reduced accordingly.

In the `BathPair.handleStratOrderAtID()`, remove the `BathToken.removeFilledTradeAmount` and only call `BathToken.removeFilledTradeAmount` in the `BathPair.rebalancePair` if tokens have been repaid back.

bghughes (Rubicon) acknowledged and commented:

> They are rebalanced in practice and the problem is it is impossible to distinguish between fill when thinking about a single strategist order. See #210

> Feature of this system to be fixed when the strategist is decentralized and no longer trusted. LMK if you want me to elaborate.

HickupHH3 (judge) decreased severity to Medium and commented:

> Issue is similar to #210 but a lot more detailed. Although they describe the problem with a slight variation, the underlying impact is the same: there will be a growing disparity between the `outstandingAmount` and `underlyingBalance` until the strategist scrubs the trade.

> Making this the primary issue.

> Copying my comment in that issue regarding lowered severity: There is a trust assumption that the strategist has to actively and frequently do scrubStrategistTrade() to ensure the accounting difference isn't large.

> It a bit of a gray area, but arguably can be viewed as a centralisation risk issue / rug vector if the strategist goes MIA. Hence, as per my reasoning outlined in #334, I'm downgrading the issue to Medium severity.

[M-16] Strategists can take more rewards than they should using the function strategistBootyClaim().

*Submitted by hansfriese, also found by oyc*109, xiaoming90, kenzo, Kumpa, unforgiven, MiloTruck, and pauliax_

Strategists can take more rewards than they should using the function strategistBootyClaim(). Even though the owner trusts strategists fully I think it’s recommended to remove such flaws.

I think there would be 2 methods to claim more rewards.

Proof of Concept

Method 1.

A strategist can call the function using same asset/quote parameters.

Then both of fillCountA and fillCountQ will be same positive values.

The first code block for fillCountA(L597-L610) will work same as expected but the second block for fillCountQ(L611-L624) will be executed for the same asset again.

Two mappings(totalFillsPerAsset, strategist2Fills) that save rewards will be updated for asset already after the first block but totalFillsPerAsset and balance of this contract for quote would be still positive as there would be remaining rewards for other strategiets.

So the strategist can get paid once more for the same asset.

Method 2.

I think a reentrancy attack is possible also because two mappings are updated after transfer funds.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

Method 1.

You can add this require() at the beginning of function.(L595) require(asset != quote, “asset = quote”);

Method 2.

You can update the state of 2 mappings before transfer. Move L608-L609 to L601 Move L622-L623 to L615

So final code will look like this.(pseudocode)

function strategistBootyClaim(address asset, address quote) external onlyApprovedStrategist(msg.sender) { require(asset != quote, “asset = quote”);

```
    uint256 fillCountA = strategist2Fills[msg.sender][asset];
    uint256 fillCountQ = strategist2Fills[msg.sender][quote];
    if (fillCountA > 0) {
        uint256 booty = (
            fillCountA.mul(IERC20(asset).balanceOf(address(this)))
        ).div(totalFillsPerAsset[asset]);

        totalFillsPerAsset[asset] -= fillCountA;
        strategist2Fills[msg.sender][asset] -= fillCountA;

        IERC20(asset).transfer(msg.sender, booty);
        emit LogStrategistRewardClaim(
            msg.sender,
            asset,
            booty,
            block.timestamp
        );
    }
    if (fillCountQ > 0) {
        uint256 booty = (
            fillCountQ.mul(IERC20(quote).balanceOf(address(this)))
        ).div(totalFillsPerAsset[quote]);

        totalFillsPerAsset[quote] -= fillCountQ;
        strategist2Fills[msg.sender][quote] -= fillCountQ;

        IERC20(quote).transfer(msg.sender, booty);
        emit LogStrategistRewardClaim(
            msg.sender,
            quote,
            booty,
            block.timestamp
        );
    }
```

```
}
```

bghughes (Rubicon) confirmed

KenzoAgada (warden) commented:

> Mentions 2 different attack paths, should probably be split: Method1 is duplicate of #238. Method2 is duplicate of #451.

HickupHH3 (judge) commented:

> I’m making this the primary issue for centralisation risks involving strategists.

> The rationale for lumping them together is that centralisation risk has been acknowledged by the sponsor, and is mentioned in their whitepaper. While this issue (and those that are grouped) mention various attack paths, they are all reliant on the strategist being malicious. The justification for the medium severity is that strategists are a larger group than the admin, so the likelihood and risk of an attack is greater.

> That being said, the recommended fixes for the issues highlighted should be adopted to make the system more trust-less and secure.

HickupHH3 (judge) commented:

> As per my reasoning in #344, this will be made the grouped primary issues for the 2 attack vectors (see rulebook’s grouping of similar issues).

[M-17] Missing checks allow strategists to steal all fund via `tailOff`

Submitted by shenwilly, also found by 0x52, Kumpa, unforgiven, pedroais, MiloTruck, and pauliax

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L533-L563

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L343-L369

Impact

Strategists can call `tailOff` with malicious payload to steal all funds within any BathToken.

There are 2 issues that makes this possible:

- `BathPair.tailOff` allows arbitrary `_stratUtil` address.
- `BathToken.rebalance` allows underlying token as `filledAssetToRebalance`.

These allow malicious strategists to input any token address, including the underlying token of a BathToken, and transfer them to a contract of their choosing.

Proof of Concept

A malicious strategist calls `tailOff` with the following payload:

```
{
    targetPool: bathUSDC.address,
    tokenToHandle: USDC.address,
    targetToken: USDT.address, // any address
    _stratUtil: maliciousContract.address,
    amount: USDC.balanceOf(bathUSDC.address),
    hurdle: 0, // any
    _poolFee: 0 // any
}
```

`bathUSDC` BathToken will then send all USDC to the strategist’s `maliciousContract`. All deposits are lost.

Recommended Mitigation Steps

- Whitelist the addresses that can be used as `_stratUtil`.
- Add a check in `rebalance` to prevent transferring underlying token: require(filledAssetToRebalance != underlyingToken, “must not be underlying”);

bghughes (Rubicon) acknowledged, but disagreed with severity

HickupHH3 (judge) decreased severity to Medium and commented:

> Making this the primary issue of strategist stealing funds via the `tailOff()` function. Rationale of med severity is outlined in #344.

bghughes (Rubicon) commented:

> Good issue, implemented the `require(filledAssetToRebalance != underlyingToken, "must not be underlying");` fix 💯

[M-18] Centralized risks allows rogue pool behavior in BathToken.

Submitted by 0x1f8b

Centralized risks allows rogue pool behavior.

Proof of Concept

The `onlyPair` modifier is as detailed below:

```
    modifier onlyPair() {
        require(
            IBathHouse(bathHouse).isApprovedPair(msg.sender) ==
            "not an approved pair - bathToken"
        );
        _;
    }
```

And the `bathHouse` is the admin as you can see in the following comment in the `initialize` method

> bathHouse = msg.sender; //NOTE: assumed admin is creator on BathHouse

So if the admin it's able to be the a valid `pair` (it could change the owner with `setBathHouse`), the owner it's able to call the method `rebalance` and steal any token.

Affected source code:

- BathToken.sol#L346-L369

Recommended Mitigation Steps

- Use timeLock, or avoid admin accounts.

bghughes (Rubicon) acknowledged and commented:

> Acknowledged centralization risk #344 #314

HickupHH3 (judge) commented:

> The attack path could have been more detailed. It is valid though:

- set bathHouse to malicious bathHouse contract to always return true for the `onlyPair` modifier check
- rug funds by specifying own wallet as destination in `rebalance()`

> It's different from #211's attack vector, hence keeping it separate.

[M-19] Strategist can transfer user funds to themselves

Submitted by Ruhum

The strategist is able to use user funds to trade on the RubiconMarket. They can abuse this to transfer user funds to themselves.

A strategist having access to user funds seems to be a deliberate design choice. But, I believe it's important to note how dangerous that is.

Proof of Concept

1. Strategist opens up an offer through placeMarketMakingTrades() where a token is sold for very cheap
2. Strategist accepts the offer within the same transaction using their private wallet

Recommended Mitigation Steps

There's no easy way to fix this since it's a big part of the protocol. You'd have to overhaul the whole thing.

You could minimize the dmg by limiting the amount of funds a strategist has access to.

bghughes (Rubicon) acknowledged and marked as duplicate:

> Duplicate of **#344**

HickupHH3 (judge) commented:

> Unique strategist rug-pull vector

[M-20] Strategists can't be removed

Submitted by kenzo, also found by Ruhum, dirky, shenwilly, and 0x1f8b_

There is no option to revoke strategist's privilege.

As the strategist is a very strategic role which can effectively steal LP's funds, this is very dangerous.

Impact

A rogue / compromised / cancelled strategist can not be revoked of permissions.

Proof of Concept

There's a function to approve a strategist, but no option to revoke the access.

Recommended Mitigation Steps

Add a function / change the function and allow setting strategist's access to false.

bghughes (Rubicon) confirmed, but disagreed with severity and commented:

> Low severity, we can add this anytime with a proxy upgrade but still a good function to add.

HickupHH3 (judge) commented:

> As per the rulebook (no. 11), upgradeability should not be used as an excuse to reduce the severity of a finding.

[M-21] User will loose funds

Submitted by csanuragjain, also found by 0x1f8b

User will lose funds if user accidentally pass route with only 1 value which is route[0]=X WETH while calling swapForETH or swapWithETH/swapEntireBalance/swap function.

Proof of Concept

1. User calls swapForETH function with below params:

```
pay_amt=500
buy_amt_min=0
route[0]=WETH
expectedMarketFeeBPS=1
```

2. User will transfer 500+fees amount to the contract

```
require(
            ERC20(route[0]).transferFrom(
                msg.sender,
                address(this),
                pay_amt.add(pay_amt.mul(expectedMarketFeeBPS).di
            ),
            "initial ERC20 transfer failed"
        );
```

3. Now _swap function is called. This function will do nothing and loop will not run due to condition failure

   ```
   for (uint256 i = 0; i < route.length - 1; i++)

   // here since route.length - 1 is 1-1=0 so loop will not run as
   ```

4. Since currentAmount will be 0 and buy*amt*min is also 0 so require(currentAmount >= buy*amt*min, “didnt clear buy*amt*min”); will pass and 0 will be returned back
5. Swap is complete and user will not receive anything

Note:

The same need to be fixed for swapWithETH,swapEntireBalance,swap function as well

Recommended Mitigation Steps

Add below check

```
require(route.length>1, "Invalid route param");
```

bghughes (Rubicon) disagreed with severity and commented:

> Medium risk because it’s user error.

pauliax (warden) commented:

> I think the problem is created by the warden by specifying `buy_amt_min=0`. If a slippage of 0 is specified then you basically anticipate that you may receive nothing. While enforcing min route length is a good suggestion, I do not think it should be that severe.

HickupHH3 (judge) decreased severity to Medium and commented:

> 2 user errors have to be made for this to happen:

- User specifies only 1 token in the route: WETH
- Zero slippage: `buy_amt_min = 0`

> Hence, because of these prerequisites, I will downgrade the issue to medium severity.

[M-22] Deprecated variables may cause DoS

Submitted by GimelSec

A malicious admin can set the deprecated variables `AqueductDistributionLive` and `AqueductAddress` to deny specific users to buy.

Proof of Concept

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconMarket.sol#L664-L669

```
            if (AqueductDistributionLive) {
                IAqueduct(AqueductAddress).distributeToMakerAndTaker
                    getOwner(id),
                    msg.sender
                );
            }
```

Admin can set the deprecated variables `AqueductDistributionLive`, `AqueductAddress` to check specific users and revert transactions to deny them.

Recommended Mitigation Steps

Delete deprecated code, but reserve variable declaration for slots if using upgradable contracts.

[bghughes (Rubicon) acknowledged, but disagreed with severity and commented](#):

> Acknowledged centralization risk [#344](#) [#314](#) [#133](#)

[HickupHH3 (judge) commented](#):

> Less so about centralization risk, more about deprecated functions. Not sure why they're kept. Makes sense to remove them (except for the storage variables of course)

> A bit of a stretch, but warden's hypothetical scenario checks out. Hence, leaving it as is.

[M-23] Possible token reentrancy in release() of BathBuddy.sol

Submitted by kebabsec, also found by kenzo, cryptphi, and 0xsomeone

If a token with callback capabilities is used as a token to vested, then a malicious beneficiary may get the vested amount back without waiting for the vesting period.

Proof of Concept

In the function release, line, there's no modifier to stop reentrancy, in the other contracts it would be the synchronized modifier. If a token could reenter with a hook in a malicious contract (an ERC777 token, for example, which is backwards compatible with ERC20), released token counter array wouldn't be updated, enabling the withdrawal of the vested amount before the vesting period ends. A plausible scenario would be:

1. A malicious beneficiary contract B calls the release() function with itself as the recipient, everything goes according to the function, and transfer and callback to the malicious beneficiary contract happens.
2. Contract B contains tokensReceived(), a function in the ERC777 token that allows for callback to the victim contract as you can see here https://twitter.com/transmissions11/status/1496944873760428058/ (This function also can be any function that is analogous to a fallback function that might be implemented in a modified ERC20. As it can be seen, any token that would give the attacker control over the execution flow will suffice.)
3. Inside the tokensReceived() function, a call is made back to the release function.
4. This steps are repeated until vested amount is taken back.
5. This allows for the malicious beneficiary contract to redeem the vested amount while bypassing the vesting period, due to the released token counter array (https://github.com/code-423n4/2022-05-

rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L116) which controls how many tokens are released (https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L101) being updated only after the transferring of all tokens occurs. As this is the case, malicious beneficiary can get the usual amount that they could withdraw at the time indefinite amount of times (as result of released in line 101 will be 0), thus approximately getting all of their vested amount back without waiting for the vesting period. (fees not included).

There's also precedents of similar bugs that reported, as seen here

Recommended Mitigation Steps

1. Consider adding a mutex such as nonReentrant, or the synchronized modifier used in the other contracts.
2. Implement checks-effects-interactions pattern.

bghughes (Rubicon) confirmed, but disagreed with severity and commented:

> This one should probably be a high priority, adding disagreement with severity.

HickupHH3 (judge) commented:

> From what I gather, `bonusTokens` can be any ERC20 token (network incentives/governance tokens), which by extension, means ERC777 tokens are possible too.

> Even if not, I suppose one could spin up a malicious bonus token to specifically target the BathBuddy contract.

> Medium severity because there isn't a loss nor stolen rewards from others, merely bypassing the vesting, which is an impact on protocol functionality.
> `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.`

[M-24] `RubiconMarket.sol#isClosed()` always returns false, making the market can not be stopped as designed

Submitted by WatchPug, also found by Ruhum, Chom, Dravee, Hawkeye, MaratCerby, minhquanym, csanuragjain, and fatherOfBlocks

```
    function isClosed() public pure returns (bool closed) {
        return false;
    }
```

> After close, no new buys are allowed.

Based on context and comments, when the market is closed, offers can only be cancelled (offer and buy will throw).

However, in the current implementation, `isClosed()` always returns `false`, so the checks on whether the market is closed will always pass. (E.g: `can_offer()`, `can_buy()`, etc)

And there is a storage variable called `stopped`, but it's never been used, which seems should be used for `isClosed`.

Recommendation

Change to:

```
    function isClosed() public pure returns (bool closed) {
        return stopped;
    }
```

bghughes (Rubicon) confirmed and commented:

> Intended functionality - confirmed

[M-25] Multiple Unsafe Arithmetic Operations

Submitted by Oxsomeone

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconMarket.sol#L844

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconMarket.sol#L857

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconMarket.sol#L883

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconMarket.sol#L898

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconMarket.sol#L927

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/RubiconMarket.sol#L951

Vulnerability details

RMT-02M: Multiple Unsafe Arithmetic Operations

File	Lines	Type
RubiconMarket.sol	L844, L857, L883, L898, L927, L951	Mathematical Operations

Description

The referenced lines all perform unsafe multiplications using the unitary denominations of either `1 ether` (`1e18`) or `10**9` (`1e9`), both of which can easily lead to overflows when used as a multiplier for large amounts of assets.

Impact

Purchasing and selling amounts will be improperly fulfilled as well as improperly tracked as “sold out” / “bought out”.

Recommended Mitigation Steps

We advise the codebase to make use of the `mul` operation exposed by the `DSMath` library already incorporated into the codebase to guarantee all operations are performed safely and cannot overflow.

Proof of Concept

Issue is deducible by inspecting the relevant lines referenced in the issue and making note of the raw multiplication (`*`) operations performed.

[bghughes (Rubicon) confirmed]

[HickupHH3 (judge) commented]:

> In order to overflow, you would need `pay_amt` to exceed `type(uint256).max / 1e18` , which would be highly unlikely with majority of the ERC20 tokens.

> Possibly arguable with ERC20 tokens of much higher decimals though. Considering that the product is an open orderbook, I'll let this issue stand.

[M-26] Malicious pools can be deployed through `BathHouse`

*Submitted by Bahurum, also found by dirky_*

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L153

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L214

Impact

Reentrancy in `BathToken.initialize()` can be exploited and this allows to create a pool which has a legitimate underlying token (even one for which a pool already exists), and has given full approval of underlying Token to an attacker. While this underlying token will differ from the one returned by `BathHouse.getBathTokenfromAsset` for that Pool (since the returned token would be the malicious one which reentered `initialize`), the LPs could still deposit actual legitimate tokens to the pool since it is deployed from the BathHouse and has the same name as a legit pool, and loose their deposit to the attacker.

Proof of Concept

Create a new pool calling `BathHouse.openBathTokenSpawnAndSignal()` and passing as `newBathTokenUnderlying` the address with the following malicious token:

```
// SPDX-License-Identifier: BUSL-1.1

pragma solidity =0.7.6;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "../../contracts/rubiconPools/BathToken.sol";

contract fakeToken is ERC20("trueToken", "TRUE"), Ownable {

    ERC20 trueToken;
    address marketAddress;
    uint256 counterApprove;
    BathToken bathToken;

    function setTrueToken(address _trueTokenAddress) onlyOwner {
        trueToken = ERC20(_trueTokenAddress);
    }

    function setMarketAddress(address _marketAddress) onlyOwner
        marketAddress = _marketAddress;
    }

    function approve(address spender, uint256 amount) public vir
        if (counterApprove == 1) { //first approve is from bathH
            bathToken = BathToken(msg.sender);
            bathToken.initialize(trueToken, owner, owner);
            attacked = false;
        }
        counterApprove++;
        _approve(_msgSender(), spender, amount);
        return true;
    }

    function setAndApproveMarket(address _market){
        // sets legitimate market after malicious bathToken init
        bathToken.setMarket(_market);
        bathToken.approveMarket();
    }

    function emptyPool() onlyOwner {
        // sends pool tokens to attacker
        uint256 poolBalance = trueToken.balanceOf(address(bathTo
        trueToken.transferFrom(address(bathToken), owner, poolBa
    }
}
```

This reenters `BathToken.initialize()` and reassigns the bathHouse role to the fake token, which names itself as the legit token. Also the reentrant call reassigns the legit Token to `underlyingToken` so thet the pool actually contains the legit token, but gives infinite approval for the legit token from the pool to the attacker, who is passed as `market` in the reentrant call.

Since the fakeToken has the bathHouse role, it can set the market to the actual RubiconMarket after the reentrant call.

Code: BathHouse.openBathTokenSpawnAndSignal, BathToken.initialize

Recommended Mitigation Steps

Add `onlyBathHouse` modifier to `initialize` function in `BathToken` to avoid reentrancy from malicious tokens.

bghughes (Rubicon) disputed and commented:

> I believe this is a bad issue for a few reasons:

- Firstly, the new bath token is *always* initialized with the implementation logic at `newBathTokenImplementation` and `initialize` is immediately called with the Bath House admin’s `RubiconMarketAddress`.
- Therefore it is impossible to call initialize twice or re-enter.
- There are system checks that treat the new token as an arbitrary ERC-20 but w/ the implementation logic and initialization flow of a BathToken (our system params_).

HickupHH3 (judge) commented:

> The POC could be made stronger with a script so that the poor judge (me) doesn't have to eyeball and manually review the attack vector. Having a working Hardhat script with the stated attack vector will 1000% strengthen the POC because it can be tested and verified.

> The issue is valid, because like #179, the key step is the re-entrancy via the approval in L214 before `initialized` is set to true.

> I think a simpler solution here is to shift the `initialized` variable to before the external call so that it follows the CEI pattern.

bghughes (Rubicon) confirmed and commented:

> Better solution and good issue warden. Thank you both.

[M-27] `RubiconMarket.feeTo` set to zero-address can DoS `buy` function

Submitted by berndartmueller, also found by 0x1f8b and blackscale

In the `RubiconMarket` contract, the `feeTo` storage variable stores the recipient of taker trade fees (i.e. `buy` function). `feeTo` can be set to any arbitrary address in `setFeeTo(address newFeeTo)`. As there is no zero-address validation, the owner can set a zero-address as the `feeTo` address.

All subsequent `buy` function calls with a `_offer.buy_gem` ERC20 token reverting on zero-address transfers (e.g. `USDC`), will revert.

Proof of Concept

RubiconMarket.sol#L297-L300

```
    require(
        _offer.buy_gem.transferFrom(msg.sender, feeTo, fee),
        "Insufficient funds to cover fee"
    );
```

Recommended Mitigation Steps

Check if `feeTo` is set to a non-zero address before transferring fees.

bghughes (Rubicon) disputed and commented:

> I do not believe having the fee recipient as the zero address would cause these transactions to revert according to EIP-20.

> Note, #344

HickupHH3 (judge) commented:

> Yes it will. USDC implementation: https://etherscan.io/address/0xa2327a938febf5fec13bacfb16ae10ecbc4cbdcf#code

```
    function _transfer(
        address from,
        address to,
        uint256 value
    ) internal override {
        require(from != address(0), "ERC20: transfer from the zero ad
        require(to != address(0), "ERC20: transfer to the zero addres
```

[M-28] RubiconRouter maxSellAllAmount does not transfer user's fund into its address, causing calls to always revert

Submitted by PP1004, also found by IllIllI

The RubiconRouter function maxSellAllAmount does not transfer user's fund into its address, causing the function to always revert

Proof of Concept

Since there is no fund transferred into router during the maxSellAllAmount call, it will always revert when RubiconMarket tries to take tokens from it.

Recommended Mitigation Steps

Add a transfer of fund to the function

```
    /// @dev this function takes a user's entire balance for the
    function maxSellAllAmount(
        ERC20 pay_gem,
        ERC20 buy_gem,
        uint256 min_fill_amount
    ) external returns (uint256 fill) {
        //swaps msg.sender entire balance in the trade
        uint256 maxAmount = ERC20(buy_gem).balanceOf(msg.sender)
        ERC20(buy_gem).safeTransferFrom(msg.sender, address(this
        fill = RubiconMarket(RubiconMarketAddress).sellAllAmount
            pay_gem,
            maxAmount,
            buy_gem,
            min_fill_amount
        );
        ERC20(buy_gem).transfer(msg.sender, fill);
    }
```

bghughes (Rubicon) confirmed

[M-29] maxSellAllAmount and maxBuyAllAmount functions can be unintentionally paused (always revert).

Submitted by PP1004, also found by hansfriese

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L290

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L307

Impact

The two functions maxSellAllAmount and maxBuyAllAmount will always revert in case at least (100-fee)% of user's balance can be matched with orders.

Proof of Concept

Let's say Bob placed an order selling 100 USDC with a low USDT price of 1:0.95.

Alice currently has 50 USDT and they want to maxSellAllAmount into USDC.

The function will pass 50 as amount into RubiconMarket's buyAll function where it fully matches with Bob's order. Here, the buy() function will first transfer alice's 50 USDT in and later 50 * feeBPS / BPS as fee. In this case, alice can not afford to pay.

Therefore, the two functions maxSellAllAmount and maxBuyAllAmount are useless in case user's request can be fully matched.

Recommended Mitigation Steps

Add the fee calculating before passing the amount to the RubiconMarket's buyAll, sellAll function.

```
/// @dev this function takes a user's entire balance for the
function maxBuyAllAmount(
    ERC20 buy_gem,
    ERC20 pay_gem,
    uint256 max_fill_amount
) external returns (uint256 fill) {
    //swaps msg.sender's entire balance in the trade

    uint256 maxAmount = _calcAmountAfterFee(ERC20(buy_gem).b

    fill = RubiconMarket(RubiconMarketAddress).buyAllAmount(
        buy_gem,
        maxAmount,
        pay_gem,
        max_fill_amount
    );
    ERC20(buy_gem).transfer(msg.sender, fill);
}
```

```
    /// @dev this function takes a user's entire balance for the
    function maxSellAllAmount(
        ERC20 pay_gem,
        ERC20 buy_gem,
        uint256 min_fill_amount
    ) external returns (uint256 fill) {
        //swaps msg.sender entire balance in the trade

        uint256 maxAmount = _calcAmountAfterFee(ERC20(buy_gem).b
        fill = RubiconMarket(RubiconMarketAddress).sellAllAmount
            pay_gem,
            maxAmount,
            buy_gem,
            min_fill_amount
        );
        ERC20(buy_gem).transfer(msg.sender, fill);
    }


    function _calcAmountAfterFee(uint256 amount) internal view r
        uint256 feeBPS = RubiconMarket(RubiconMarketAddress).get
        return amount.sub(amount.mul(feeBPS).div(10000));
    }
```

bghughes (Rubicon) confirmed

HickupHH3 (judge) commented:

> Realised this is a separate issue from not transferring funds into the router. #376 will be the primary issue for that issue.

[M-30] `BathBuddy` contract's `vestedAmount` function includes fees leading to users being disproportionately rewarded after whale withdraws

Submitted by sseefried

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L103-L104 https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L133

Impact

When a whale withdraws their tokens and receives rewards from the `BathBuddy` contract the fees they pay will erroneously become part of the calculation performed in function `vestedAmount`. This means that any subsequent withdrawer of funds may receive a disproportionate amount of tokens. The fees paid by a whale could still be much larger than the amount of tokens invested by a minnow.

Although similar to the issue “When `BathToken` contract is recipient of fees then users can make disproportionate returns after whales withdraw” it is not the same issue since fees are always accrued in the `BathBuddy` contract and this cannot be changed. Also, the calculations in are subtly different

However, the outcome is the same. A minnow can receive a disproportionate reward and drain much of the fees from the contract.

The intention of setting the pool as the recipient of the fees was to reward HODLers but, in fact, they will be incentivised to withdraw after a whale does.

Proof of Concept

Consider the following scenario.

1. fee is set to 50 BPS (i.e. 0.50%)
2. A whale deposits 200 tokens
3. A minnow deposits 0.01 tokens
4. A `BathBuddy` contract is set up for the `BathToken` contract.
5. The whale withdraws their funds
6. The minnow then withdraws their funds

After step 5, the function `vestedAmount` will return a value that includes the fees paid by the whale. This is because the `BathBuddy` contract is the recipient of all fees. They are not transferred anywhere.

Thus, when the minnow withdraws their funds `releasable` is much larger than the amount they otherwise would have expected. Further sharesWithdrawn is equal to initialTotalSupply in this particular scenario so `mul(sharesWithdrawn).div(initialTotalSupply)` evaluates to `1`. This means that `amount = releaseable`.

A test has been written in the private fork that exhibits this behaviour.

Recommended Mitigation Steps

Keep a tally of the fees accrued in a separate variable and work out a fairer system for distributing rewards to HODLers.

csanuragjain (warden) commented:

> The totalAllocation is set as IERC20(token).balanceOf(address(this)) + released(token)

> In this case

> a. IERC20(token).balanceOf(address(this)) includes the fees as warden mentioned

> b. released(token) also includes the fees since _erc20Released[address(token)] is updated with amount and not amountWithdrawn BathBuddy.sol#L116

bghughes (Rubicon) disputed and commented:

> I believe this is a non-issue and disagree with the assessment for one key reason: the fees accrued still adhere to the vesting schedule. You are correct that after a whale withdraws there are then more rewards to be withdrawn - that's a feature, not a bug! Importantly, that fee accrual only adds to the pool stack, and although true that the minnow can withdraw right now for marginal gains (their share of the vested fee amount, fees vest as if vesting from the start) they have no reason to do so because they pay the same withdrawal fee.

> What do you think of this situation @csanuragjain ? Would love another opinion :)

csanuragjain (warden) commented:

> @bghughes I agree with your point but here fees is added twice while calculating the totalAllocation. totalAllocation is calculated as IERC20(token).balanceOf(address(this)) + released(token).

> The problem here is both IERC20(token).balanceOf(address(this)) and released(token) are inclusive of the fees which makes double sum of fees

> a. IERC20(token).balanceOf(address(this)) includes the fees as all fees are kept in contract itself b. released(token) also includes the fees since _erc20Released[address(token)] is updated with amount and not amountWithdrawn BathBuddy.sol#L116

> Since totalAllocation represent a balance which is not even present in the contract (increased by fee amount twice), so contract wont have enough fund to send reward

> Can you please suggest

HickupHH3 (judge) commented:

> I agree with @csanuragjain: there seems to be a double increment of the fees.

- `_erc20Released[token]` is incremented by `amount`, which consists of the fee
- The fee is kept within the contract, not transferred out => `balanceOf()` consists of fee too
- > Calculated vested amount

  ```
  IERC20(token).balanceOf(address(this)) + released(token),
  ```

  > therefore is a double fee accounting.

> Seems like a high-severity bug that no other wardens caught o.O

HickupHH3 (judge) commented:

> Anyway, regarding the main issue, because no funds are stolen or compromised per se, and can be treated as rewards / incentives, I'll be downgrading it to medium as it

> relates to reward distribution that the remaining stakers aren't entitled to yet (and thus isn't considered to be "stolen" or "compromised").

> As leastwood puts it: "Protocol leaked value has a broad context but I think most judges can agree that it would pertain to rewards being paid out a lower rate than expected. Or, users can extract small amounts (up to debate on what is considered to be small) from the protocol under certain assumptions. "

bghughes (Rubicon) confirmed and commented:

> Great, thank you @HickupHH3

[M-31] Lack of Access Control for offer(uint, ERC20, uint, ERC20) and insert(uint, unint)

Submitted by xiaoming90, also found by throttle

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L598

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L697

Proof of Concept

The `offer(uint, ERC20, uint, ERC20)` and `insert(uint, unint)` should only be accessible by the keepers as per the comments. However, there is no authorisation logic or access control implemented. Therefore, anyone could call these two functions.

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L598

```
    // Make a new offer. Takes funds from the caller into market esc
    //
    // If matching is enabled:
```

```
    //     * creates new offer without putting it in
    //       the sorted list.
    //     * available to authorized contracts only!
    //     * keepers should call insert(id,pos)
    //       to put offer in the sorted list.
    //
    // If matching is disabled:
    //     * calls expiring market's offer().
    //     * available to everyone without authorization.
    //     * no sorting is done.
    //
    function offer(
        uint256 pay_amt, //maker (ask) sell how much
        ERC20 pay_gem, //maker (ask) sell which token
        uint256 buy_amt, //taker (ask) buy how much
        ERC20 buy_gem //taker (ask) buy which token
    ) public override returns (uint256) {
        require(!locked, "Reentrancy attempt");


            function(uint256, ERC20, uint256, ERC20) returns (uint25
         = matchingEnabled ? _offeru : super.offer;
        return fn(pay_amt, pay_gem, buy_amt, buy_gem);
    }
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L697

```
    //insert offer into the sorted list
    //keepers need to use this function
    function insert(
        uint256 id, //maker (ask) id
        uint256 pos //position to insert into
    ) public returns (bool) {
        require(!locked, "Reentrancy attempt");
        require(!isOfferSorted(id)); //make sure offers[id] is not y
        require(isActive(id)); //make sure offers[id] is active

        _hide(id); //remove offer from unsorted offers list
        _sort(id, pos); //put offer into the sorted offers list
        emit LogInsert(msg.sender, id);
        return true;
        }
```

Impact

Following are the three offers functions that public users can use to place new orders

- offer(uint, ERC20, uint, ERC20, uint)
- offer(uint, ERC20, uint, ERC20, uint, bool)
- offer(uint, ERC20, uint, ERC20)

Per the OasisDex Documentation, which Rubicon Market based upon, the last order (`offer(uint, ERC20, uint, ERC20)`) method should not be used. Following is the extract from the documentation.

> This method IS NOT recommended and shouldn't be used. Such an offer would not end up in the sorted list but would rather need to be inserted by a keeper at a later date. There is no guarantee that this will ever happen.

`offer(uint, ERC20, uint, ERC20)` and `insert(uint, unint)` should be reserved for authorized users (e.g. keepers) only, but the fact is that anyone could access.

The functions `offer(uint,ERC20,uint,ERC20,uint)` and `offer(uint,ERC20,uint,ERC20,uint,bool)` will trigger the matching logic, but the function `offer(uint,ERC20,uint,ERC20)` does not.

The function `offer(uint,ERC20,uint,ERC20)` allows malicious user to manipulate the orderbook in an atomic transaction by submitting a order without it being atomically matched, and then `insert(uint,uint)` can be used in order to manually sort the order without triggering matching.

These additional interfaces might potentially allow attacker to implement sophisticated techniques to compromise the protocol in the future. These two interfaces have been utilised by malicious users in the past to manipulate the orderbook, see https://samczsun.com/taking-undercollateralized-loans-for-fun-and-for-profit/ (Eth2Dai Section)

Recommended Mitigation Steps

Review if `offer(uint, ERC20, uint, ERC20)` and `insert(uint, unint)` is needed. If these function are not needed, it is recommended to remove these functions to reduce the attack surface of the protocol. If these functions are needed, implement the necessary access controls to ensure only authorised users can access.

bghughes (Rubicon) disputed and commented:

> I believe this should be informational as it is a feature to allow for users to create offers outside of the sorted list. Them then `inserting` that offer into the list seems like appropriate functionality to me.

HickupHH3 (judge) commented:

> The warden has referenced a past attack vector demonstrated by the legendary samczsun that exploited the exact same functions to manipulate prices, as well as OasisDex's documentation, which makes the issue a very strong case.

> Have to therefore disagree that it's appropriate functionality. The functions mentioned by the warden should be removed to prevent potential integrations from being exploited the same way.

[M-32] Changing `matchingEnabled` in `RubiconMarket` breaks protocol

Submitted by berndartmueller

If `matchingEnabled` in `RubiconMarket` is changed from `false` to `true`, all offers created while `matchingEnabled = false` can not be matched or canceled if `matchingEnabled` is toggled to `true` again. Only changing the value `matchingEnabled` back to `false` allows those offers to be used again.

Proof of Concept

Offers created while `matchingEnabled = false` do not populate `_rank` and `_best`. However, both variables are used in multiple functions `isOfferSorted`, `_unsort`, `_hide` which are used if `matchingEnabled = true`.

For example, canceling an offer that has been created when `matchingEnabled = false` and enabling matching again, will revert:

RubiconMarket.sol#L685-L691

```
    if (matchingEnabled) {
        if (isOfferSorted(id)) {
            require(_unsort(id)); // @audit-info will revert for offe
        } else {
            require(_hide(id)); // @audit-info will revert for offer
        }
    }
```

Recommended Mitigation Steps

Consider removing the functionality to toggle `matchingEnabled` or only allow changing `matchingEnabled` to `false` without changing the value back to `true`.

bghughes (Rubicon) disputed and commented:

> This seems like an edge case to me, this kind of toggle could still be permissible in practice but we intend to only keep our value at `true`.

HickupHH3 (judge) commented:

> Valid issue, even if it's an edge case. As auditors, between what the code can actually do (and it's impact) versus what it's intended to, the former is arguably more important.

[M-33] RubiconMarketAddress in BathPair can't be updated

Submitted by pauliax

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/rubiconPools/BathPair.sol#L128

https://github.com/code-423n4/2022-05-rubicon/blob/main/contracts/rubiconPools/BathHouse.sol#L335-L337

Impact

RubiconMarketAddress in BathPair is initialized only once:

```
    RubiconMarketAddress = IBathHouse(_bathHouse).getMarket();
```

but market can change in Bath house:

```
        /// @notice Admin-only function to set a Bath Token's target
        function setMarket(address newMarket) external onlyAdmin {
            RubiconMarketAddress = newMarket;
        }
```

Thus it will get out of sync. Also, the comment says that it changes Bath Token's target Rubicon market but actually it updates its own instance variable.

Recommended Mitigation Steps

I think RubiconMarketAddress should sync between BathPair and BathHouse.

bghughes (Rubicon) acknowledged and commented:

> Not needed in practice.

HickupHH3 (judge) commented:

> Issue stands because there's a non-zero possibility of it happening.

[M-34] `RubiconMarket` buys can not be disabled if offer matching is disabled

Submitted by berndartmueller

In the `RubiconMarket` contract, buys can be disabled with `setBuyEnabled`. However, if `matchingEnabled` is set to `false`, buys can not be disabled as the `require` check is located in the `_buys` function instead of checking `buyEnabled` in the `buy` function.

Proof of Concept

[RubiconMarket.sol#L962]()

```
    function _buys(uint256 id, uint256 amount) internal returns (boo
        require(buyEnabled); // @audit-info Buys can not be disabled
        if (amount == offers[id].pay_amt) {
            if (isOfferSorted(id)) {
                //offers[id] must be removed from sorted list becaus
                _unsort(id);
            } else {
                _hide(id);
            }
        }

        require(super.buy(id, amount));

        // If offer has become dust during buy, we cancel it
        if (
            isActive(id) &&
            offers[id].pay_amt < _dust[address(offers[id].pay_gem)]
        ) {
            dustId = id; //enable current msg.sender to call cancel(
            cancel(id);
        }
        return true;
    }
```

Recommended Mitigation Steps

Move the `require` check for `buyEnabled` to the `buy` function here.

bghughes (Rubicon) disputed and commented:

> Seems irrelevant as `buyEnabled` is working as intended.

HickupHH3 (judge) commented:

> @bghughes kindly double check; the warden is correct.

> If `matchingEnabled` is false, then it will enter the latter case of the ternary statement:

```
function(uint256, uint256) returns (bool) fn = matchingEnabled
    ? _buys
    : super.buy;
```

> https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L670-L672

> which means buys can still be executed because the `buyEnabled` isn't checked in the parent function. This probably stems from the intended functionality of the `buyEnabled` variable: is it supposed to disable entirely, or only when matching is enabled?
> As the warden's concern is valid, I'll be letting the issue stand.

bghughes (Rubicon) confirmed and commented:

> Thank you @HickupHH3 for your support throughout the review process and apologies for being late to reply. You are correct, it seems this is a valid issue, I was dismissive because we never use this in practice, always having `matchingEnabled` and buying enabled.

Low Risk and Non-Critical Issues

For this contest, 72 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by IllIllI received the top score from the judge.

The following wardens also submitted reports: berndartmueller, 0x1f8b, fatherOfBlocks, Dravee, horsefacts, Bahurum, pauliax, unforgiven, rotcivegaf, joestakey, sashik_eth, blockdev, sorrynotsorry, sseefried, 0xNazgul, hansfriese, hubble, gzeon, defsec, PP1004, csanuragjain, MaratCerby, TerrierLover, ellahi, Funen, ilan, simon135, GimelSec, Hawkeye, StErMi, xiaoming90, SmartSek, oyc_109, 0xDjango, 0xf15ers, blackscale, eccentricexit, minhquanym, Waze, c3phas, catchup, Ruhum, shenwilly, _Adam, 0x4non, broccolirob, CertoraInc, sach1r0, FSchmoede, WatchPug, delfin454000, Metatron, BouSalman, JMukesh, Picodes, 0xKitsune, ACai, Chom, cryptphi, ElKu, throttle, UnusualTurtle, UVvirus, Kaiziron, kebabsec, 0x1337, AlleyCat, asutorufos, dipp, JC, *and* parashar.

Low Risk Issues

	Issue	Instances
1	Unused/empty `receive()` / `fallback()` function	3
2	Return unused fees	5
3	Migrations should do some validation	1
4	Front-runable initializer	1
5	Vulnerable to cross-chain replay attacks due to static DOMAIN_SEPARATOR	1
6	Contracts should extend interfaces they extend	1
7	NatSpec incorrect	1
8	Misleading function name	1
9	Missing checks for `address(0x0)` when assigning values to `address` state variables	16

Total: 30 instances over 9 issues

[L-01] Unused/empty `receive()` / `fallback()` function

If the intention is for the Ether to be used, the function should call another function, otherwise it should revert (e.g. `require(msg.sender == address(weth))`)

There are 3 instances of this issue:

```
File: contracts/peripheral_contracts/BathBuddy.sol   #1

69:        receive() external payable {}
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L69

```
File: contracts/RubiconRouter.sol   #2

37:       receive() external payable {}
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L37

```
File: contracts/RubiconRouter.sol   #3

39:       fallback() external payable {}
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L39

[L-02] Return unused fees

The `expectedMarketFeeBPS` argument to a bunch of functions is a user projection. The actual fee is a static value set by the admin, not a dynamic one, so the difference between the two values should be returned, as is done for some of the other token transfers in this contract.

There are 5 instances of this issue:

```
File: contracts/RubiconRouter.sol

194      function swap(
195          uint256 pay_amt,
196          uint256 buy_amt_min,
197          address[] calldata route, // First address is what .
198          uint256 expectedMarketFeeBPS //20
199:     ) public returns (uint256) {

267      function swapEntireBalance(
268          uint256 buy_amt_min,
269          address[] calldata route, // First address is what .
270          uint256 expectedMarketFeeBPS
271:     ) external returns (uint256) {
```

```
function buyAllAmountWithETH(
    ERC20 buy_gem,
    uint256 buy_amt,
    uint256 max_fill_amount,
    uint256 expectedMarketFeeBPS
) external payable returns (uint256 fill) {

function swapWithETH(
    uint256 pay_amt,
    uint256 buy_amt_min,
    address[] calldata route, // First address is what
    uint256 expectedMarketFeeBPS
) external payable returns (uint256) {

function swapForETH(
    uint256 pay_amt,
    uint256 buy_amt_min,
    address[] calldata route, // First address is what
    uint256 expectedMarketFeeBPS
) external payable returns (uint256 fill) {
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L194-L199

[L-03] Migrations should do some validation

At the very least, a new bath token should have the same underlying to make sure funds can be taken out

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathHouse.sol   #1

/// @notice A migration function that allows the admin
function adminWriteBathToken(ERC20 overwriteERC20, addre
    external
    onlyAdmin
{
    tokenToBathToken[address(overwriteERC20)] = newBath
    emit LogNewBathToken(
        address(overwriteERC20),
224                 newBathToken,
225                 address(0),
226                 block.timestamp,
227                 msg.sender
228             );
229:    }
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L216-L229

[L-04] Front-runable initializer

There is nothing preventing another account from calling the initializer before the contract owner. In the best case, the owner is forced to waste gas and re-deploy. In the worst case, the owner does not notice that his/her call reverts, and everyone starts using a contract under the control of an attacker

There is 1 instance of this issue:

```
File: contracts/RubiconRouter.sol   #1

41:      function startErUp(address _theTrap, address payable _w
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L41

[L-05] Vulnerable to cross-chain replay attacks due to static DOMAIN_SEPARATOR

See this issue from a prior contest for details

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1

199             DOMAIN_SEPARATOR = keccak256(
200                 abi.encode(
201                     keccak256(
202                         "EIP712Domain(string name,string versio
203                     ),
204                     keccak256(bytes(name)),
205                     keccak256(bytes("1")),
206                     chainId,
207                     address(this)
208:                )
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L199-L208

[L-06] Contracts should extend interfaces they extend

Extending an interface ensures that all function signatures are correct, and catches mistakes introduced (e.g. through errant keystrokes)

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1

387:     /// @notice * EIP 4626 *
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L387

[L-07] NatSpec incorrect

The NatSpec does not match what is being done by the function

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathHouse.sol   #1

285:     /// @notice Admin-only function to set a Bath Token's t
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L285

[L-08] Misleading function name

`setBonusToken()` is actually appending a new bonus token to the previous list of bonus tokens. The function name should reflect the action it's taking, or else reviewers and users will get confused about what the code is actually doing

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1

    /// @notice Admin-only function to add a bonus token to
    function setBonusToken(address newBonusERC20) external
        bonusTokens.push(newBonusERC20);
    }
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L269-L272

[L-09] Missing checks for `address(0x0)` when assigning values to `address` state variables

There are 16 instances of this issue:

See original submission for details.

Non-Critical Issues

	Issue	Instances
1	Consider addings checks for signature malleability	1
2	No use of two-phase ownership transfers	1
3	Return values of `approve()` not checked	3

	Issue	Instances
4	Adding a `return` statement when the function defines a named return variable, is redundant	2
5	`require()` / `revert()` statements should have descriptive reason strings	55
6	`public` functions not called by the contract should be declared `external` instead	26
7	Non-assembly method available	1
8	`2**<n> - 1` should be re-written as `type(uint<n>).max`	5
9	`type(uint<n>).max` should be used instead of `uint<n>(-1)`	1
10	`constant`s should be defined rather than using magic numbers	43
11	Redundant cast	1
12	Missing event for critical parameter change	13
13	Use a more recent version of solidity	2
14	Use a more recent version of solidity	3
15	Use scientific notation (e.g. `1e18`) rather than exponentiation (e.g. `10**18`)	10
16	Inconsistent spacing in comments	127
17	Variable names that consist of all capital letters should be reserved for `const` / `immutable` variables	1
18	Typos	17
19	NatSpec is incomplete	1
20	Event is missing `indexed` fields	33
21	Not using the named return variables anywhere in the function is confusing	15

Total: 361 instances over 21 issues

[N-01] Consider addings checks for signature malleability

Use OpenZeppelin’s `ECDSA` contract rather than calling `ecrecover()` directly

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1

739:         address recoveredAddress = ecrecover(digest, v, r, 
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L739

[N-02] No use of two-phase ownership transfers

Consider adding a two-phase transfer, where the current owner nominates the next owner, and the next owner has to call `accept*()` to become the new owner. This prevents passing the ownership to an account that is unable to use it.

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathHouse.sol   #1

253      function setBathHouseAdmin(address newAdmin) external o
254          admin = newAdmin;
255:     }
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L253-L255

[N-03] Return values of `approve()` not checked

Not all `IERC20` implementations `revert()` when there's a failure in `approve()`. The function signature has a `boolean` return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually approving anything

There are 3 instances of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1
```

```
214:            IERC20(address(token)).approve(RubiconMarketAddres
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L214

```
File: contracts/rubiconPools/BathToken.sol   #2

256:            underlyingToken.approve(RubiconMarketAddress, 2**2
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L256

```
File: contracts/RubiconRouter.sol   #3

465:                target.approve(targetPool, amount);
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L465

[N-04] Adding a `return` statement when the function defines a named return variable, is redundant

There are 2 instances of this issue:

```
File: contracts/rubiconPools/BathPair.sol   #1

315:                    return _index;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L315

```
File: contracts/RubiconRouter.sol   #2

379:          return fill;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L379

[N-05] `require()` / `revert()` statements should have descriptive reason strings

There are 55 instances of this issue:

See original submission for details.

[N-06] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

There are 26 instances of this issue:

See original submission for details.

[N-07] Non-assembly method available

`assembly{ id := chainid() }` => `uint256 id = block.chainid;`

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1

197:              chainId := chainid()
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L197

[N-08] `2**<n> - 1` should be re-written as `type(uint<n>).max`

Earlier versions of solidity can use `uint<n>(-1)` instead. Expressions not including the `- 1` can often be re-written to accomodate the change (e.g. by using a `>` rather than a `>=`, which will also save some gas)

There are 5 instances of this issue:

```
File: contracts/rubiconPools/BathToken.sol

214:            IERC20(address(token)).approve(RubiconMarketAddres

256:            underlyingToken.approve(RubiconMarketAddress, 2**2

421:            maxAssets = 2**256 - 1; // No limit on deposits in

451:            maxShares = 2**256 - 1; // No limit on shares that
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L214

```
File: contracts/RubiconRouter.sol

157:            ERC20(toApprove).approve(RubiconMarketAddress, 2**
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L157

[N-09] `type(uint<n>).max` should be used instead of `uint<n>(-1)`

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1

703:          if (allowance[from][msg.sender] != uint256(-1)) {
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L703

[N-10] `constant` s should be defined rather than using magic numbers

There are 43 instances of this issue:

See original submission for details.

[N-11] Redundant cast

The type of the variable is the same as the type to which the variable is being cast

There is 1 instance of this issue:

```
File: contracts/RubiconRouter.sol   #1

/// @audit address(wethAddress)
331:          address _weth = address(wethAddress);
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L331

[N-12] Missing event for critical parameter change

There are 13 instances of this issue:

See original submission for details.

[N-13] Use a more recent version of solidity

Use a solidity version of at least 0.8.13 to get the ability to use `using for` with a list of free functions

There are 2 instances of this issue:

```
File: contracts/rubiconPools/BathPair.sol   #1

8:    pragma solidity =0.7.6;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L8

```
File: contracts/peripheral_contracts/BathBuddy.sol   #2

2:    pragma solidity >=0.6.0 <0.8.0;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L2

[N-14] Use a more recent version of solidity

Use a solidity version of at least 0.8.4 to get `bytes.concat()` instead of `abi.encodePacked(<bytes>,<bytes>)` Use a solidity version of at least 0.8.12 to get `string.concat()` instead of `abi.encodePacked(<str>,<str>)`

There are 3 instances of this issue:

```
File: contracts/RubiconMarket.sol   #1
```

```
7:    pragma solidity =0.7.6;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L7

```
File: contracts/rubiconPools/BathToken.sol   #2

8:    pragma solidity =0.7.6;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L8

```
File: contracts/RubiconRouter.sol   #3

5:    pragma solidity =0.7.6;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L5

[N-15] Use scientific notation (e.g. `1e18`) rather than exponentiation (e.g. `10**18`)

There are 10 instances of this issue:

```
File: contracts/RubiconMarket.sol

73:        uint256 constant WAD = 10**18;

74:        uint256 constant RAY = 10**27;

857:                            pay_amt * 10**9,

859:                        ) / 10**9;
```

```
898:                            buy_amt * 10**9,

900:                        ) / 10**9

927:                    pay_amt * 10**9,

929:                ) / 10**9

951:                    buy_amt * 10**9,

953:                ) / 10**9
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L73

[N-16] Inconsistent spacing in comments

Some lines use `// x` and some use `//x`. The instances below point out the usages that don't follow the majority, within each file.

There are 127 instances of this issue:

See original submission for details.

[N-17] Variable names that consist of all capital letters should be reserved for `const` / `immutable` variables

If the variable needs to be different based on which class it comes from, a `view` / `pure` *function* should be used instead (e.g. like this).

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1

66:       bytes32 public DOMAIN_SEPARATOR;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L66

[N-18] Typos

There are 17 instances of this issue:

See original submission for details.

[N-19] NatSpec is incomplete

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathHouse.sol   #1

/// @audit Missing: '@return'
          /// @param underlyingERC20 The underlying ERC-20 asset
          /// @param _feeAdmin Recipient of pool withdrawal fees
          function _createBathToken(ERC20 underlyingERC20, addre
              internal
              returns (address newBathTokenAddress)
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L389-L393

[N-20] Event is missing `indexed` fields

Each `event` should use three `indexed` fields if there are three or more fields

There are 33 instances of this issue:

See original submission for details.

[N-21] Not using the named return variables anywhere in the function is confusing

Consider changing the variable to be an unnamed one

There are 15 instances of this issue:

See original submission for details.

Gas Optimizations

For this contest, 57 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IllIllI received the top score from the judge.

The following wardens also submitted reports: sashik_eth, pauliax, 0x1f8b, joestakey, Dravee, 0xkatana, gzeon, MaratCerby, Waze, defsec, fatherOfBlocks, MiloTruck, c3phas, Tomio, _Adam, oyc_109, ElKu, rotcivegaf, simon135, 0xDjango, blockdev, csanuragjain, ellahi, FSchmoede, blackscale, delfin454000, hansfriese, Kaiziron, reassor, 0xf15ers, 0xNazgul, catchup, DavidGialdi, ilan, Metatron, UnusualTurtle, Funen, minhquanym, SmartSek, WatchPug, rfa, 0x4non, pedroais, berndartmueller, Randyyy, sach1r0, antonttc, Chom, samruna, z3s, Picodes, asutorufos, Fitraldys, GimelSec, JC, *and* RoiEvenHaim.

	Issue	Instances
1	Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate	2
2	State variables only set in the constructor should be declared `immutable`	3
3	State variables can be packed into fewer storage slots	3
4	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	12
5	Avoid contract existence checks by using solidity version 0.8.10 or later	117
6	State variables should be cached in stack variables rather than re-reading them from storage	71
7	Multiple accesses of a mapping should use a local variable cache	59
8	`internal` functions only called once can be inlined to save gas	12
9	`<array>.length` should not be looked up in every loop of a `for`-loop	3
10	`require()` / `revert()` strings longer than 32 bytes cost extra gas	26
11	`keccak256()` should only need to be called on a specific string literal once	1

	Issue	Instances
12	Using `bool` s for storage incurs overhead	12
13	Use a more recent version of solidity	5
14	Use a more recent version of solidity	1
15	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	6
16	It costs more gas to initialize variables to zero than to let the default of zero be applied	13
17	`internal` functions not called by the contract should be removed to save deployment gas	7
18	`++i` costs less gas than `i++` , especially when it's used in `for` -loops (`--i` / `i--` too)	8
19	Splitting `require()` statements that use `&&` saves gas	8
20	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	37
21	Using `private` rather than `public` for constants, saves gas	1
22	Don't compare boolean expressions to boolean literals	4
23	Duplicated `require()` / `revert()` checks should be refactored to a modifier or function	12
24	Division by two should use bit shifting	4
25	`require()` or `revert()` statements that check input arguments should be at the top of the function	4
26	Empty blocks should be removed or emit something	3
27	Superfluous event fields	8
28	Functions guaranteed to revert when called by normal users can be marked `payable`	35

Total: 477 instances over 28 issues

[1] Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate

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.

There are 2 instances of this issue:

```
File: contracts/RubiconMarket.sol   #1

543       mapping(address => mapping(address => uint256)) public
544       mapping(address => mapping(address => uint256)) public
545:      mapping(address => uint256) public _dust; //minimum se
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L543-L545

```
File: contracts/rubiconPools/BathPair.sol   #2

52        mapping(address => mapping(address => mapping(address =
53            public outOffersByStrategist;
54
55        /// @notice Tracks the market-kaing fill amounts on a
56        /// @dev strategist => erc20asset => fill amount per a
57:       mapping(address => mapping(address => uint256)) public
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L52-L57

[2] State variables only set in the constructor should be declared `immutable`

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

There are 3 instances of this issue:

```
File: contracts/peripheral_contracts/BathBuddy.sol   #1

31:       address public beneficiary;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L31

```
File: contracts/peripheral_contracts/BathBuddy.sol   #2

32:       uint64 public start;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L32

```
File: contracts/peripheral_contracts/BathBuddy.sol   #3

33:       uint64 public duration;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L33

[3] State variables can be packed into fewer storage slots

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

There are 3 instances of this issue:

```
File: contracts/RubiconMarket.sol   #1

/// @audit Variable ordering with 4 slots instead of the current
/// @audit uint256(32):last_offer_id, mapping(32):offers, uint25
186:      uint256 public last_offer_id;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L186

```
File: contracts/rubiconPools/BathHouse.sol   #2

/// @audit Variable ordering with 10 slots instead of the curren
/// @audit string(32):name, mapping(32):approvedStrategists, uin
20:      string public name;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L20

```
File: contracts/rubiconPools/BathToken.sol   #3

/// @audit Variable ordering with 17 slots instead of the curren
/// @audit string(32):symbol, string(32):name, uint256(32):feeBP
22:      bool public initialized;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L22

[4] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 * <mem_array>.length`).

Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one

There are 12 instances of this issue:

```
File: contracts/rubiconPools/BathPair.sol

413:            address[2] memory tokenPair, // ASSET, Then Quote

414:            uint256[] memory askNumerators, // Quote / Asset

415:            uint256[] memory askDenominators, // Asset / Quote

416:            uint256[] memory bidNumerators, // size in ASSET

417:            uint256[] memory bidDenominators // size in QUOTES

464:            uint256[] memory ids,

465:            address[2] memory tokenPair, // ASSET, Then Quote

466:            uint256[] memory askNumerators, // Quote / Asset

467:            uint256[] memory askDenominators, // Asset / Quote

468:            uint256[] memory bidNumerators, // size in ASSET

469:            uint256[] memory bidDenominators // size in QUOTES

578:        function scrubStrategistTrades(uint256[] memory ids)
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L413

[5] Avoid contract existence checks by using solidity version 0.8.10 or later

Prior to 0.8.10 the compiler inserted extra code, including `EXTCODESIZE` (**700 gas**), to check for contract existence for external calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value.

There are 117 instances of this issue:

See original submission for details.

[6] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replace each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 71 instances of this issue:

See original submission for details.

[7] Multiple accesses of a mapping should use a local variable cache

The instances below point to the second+ access of a value inside a mapping, within a function. Caching a mapping's value in a local `storage` variable when the value is accessed multiple times, saves **~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.

There are 59 instances of this issue:

See original submission for details.

[8] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 12 instances of this issue:

```
File: contracts/RubiconMarket.sol

32:       function isAuthorized(address src) internal view retur

435:      function _next_id() internal returns (uint256) {

1001:     function _findpos(uint256 id, uint256 pos) internal vi

1049      function _matcho(
1050          uint256 t_pay_amt, //taker sell how much
1051          ERC20 t_pay_gem, //taker sell which token
1052          uint256 t_buy_amt, //taker buy how much
1053          ERC20 t_buy_gem, //taker buy which token
1054          uint256 pos, //position id
1055          bool rounding //match "close enough" orders?
1056:     ) internal returns (uint256 id) {
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L32

```
File: contracts/rubiconPools/BathPair.sol

160       function enforceReserveRatio(
161           address underlyingAsset,
162           address underlyingQuote
163       )
164           internal
165           view
166:          returns (address bathAssetAddress, address bathQuo

205:      function _next_id() internal returns (uint256) {

213:      function handleStratOrderAtID(uint256 id) internal {
```

```
305        function getIndexFromElement(uint256 uid, uint256[] st
306            internal
307            view
308:           returns (uint256 _index)
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L160-L166

```
File: contracts/rubiconPools/BathToken.sol

629        function distributeBonusTokenRewards(
630            address receiver,
631            uint256 sharesWithdrawn,
632:           uint256 initialTotalSupply

657:       function _mint(address to, uint256 value) internal {

663:       function _burn(address from, uint256 value) internal {
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L629-L632

```
File: contracts/peripheral_contracts/BathBuddy.sol

149        function _vestingSchedule(uint256 totalAllocation, uin
150            internal
151            view
152:           returns (uint256)
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L149-L152

[9] `<array>.length` should not be looked up in every loop of a `for`-loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)
- calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset

There are 3 instances of this issue:

```
File: contracts/rubiconPools/BathPair.sol    #1

311:            for (uint256 index = 0; index < array.length; inde
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L311

```
File: contracts/rubiconPools/BathPair.sol    #2

582:            for (uint256 index = 0; index < ids.length; index+
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L582

```
File: contracts/rubiconPools/BathToken.sol    #3

635:                for (uint256 index = 0; index < bonusTokens.le
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L635

[10] `require()` / `revert()` strings longer than 32 bytes cost extra gas

There are 26 instances of this issue:

[See original submission](#) for details.

[11] `keccak256()` should only need to be called on a specific string literal once

It should be saved to an immutable variable, and the variable used instead. If the hash is being used as a part of a function selector, the cast to `bytes4` should also only be done once

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1

201                       keccak256(
202                           "EIP712Domain(string name,string versi
203:                      ),
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L201-L203

[12] Using `bool` s for storage incurs overhead

```
    // Booleans are more expensive than uint256 or any type that
    // word because each write operation emits an extra SLOAD to
    // slot's contents, replace the bits taken up by the boolean
    // back. This is the compiler's defense against contract upg
    // pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27 Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess ([100 gas](https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L191)), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past

There are 12 instances of this issue:

```
File: contracts/RubiconMarket.sol

191:       bool locked;

449:       bool public stopped;

526:       bool public buyEnabled = true; //buy enabled

527:       bool public matchingEnabled = true; //true: enable mat

530:       bool public initialized;

533:       bool public AqueductDistributionLive;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L191

```
File: contracts/rubiconPools/BathHouse.sol

32:       mapping(address => bool) public approvedStrategists;

35:       bool public initialized;

38:       bool public permissionedStrategists;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L32

```
File: contracts/rubiconPools/BathPair.sol

32:       bool public initialized;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L32

```
File: contracts/rubiconPools/BathToken.sol

22:       bool public initialized;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L22

```
File: contracts/RubiconRouter.sol

23:       bool public started;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L23

[13] Use a more recent version of solidity

Use a solidity version of at least 0.8.0 to get overflow protection without `SafeMath` Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There are 5 instances of this issue:

```
File: contracts/RubiconMarket.sol

7:    pragma solidity =0.7.6;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L7

```
File: contracts/rubiconPools/BathHouse.sol

7:    pragma solidity =0.7.6;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L7

```
File: contracts/rubiconPools/BathPair.sol

8:    pragma solidity =0.7.6;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L8

```
File: contracts/rubiconPools/BathToken.sol

8:    pragma solidity =0.7.6;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L8

```
File: contracts/RubiconRouter.sol

5:    pragma solidity =0.7.6;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L5

[14] Use a more recent version of solidity

Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There is 1 instance of this issue:

```
File: contracts/peripheral_contracts/BathBuddy.sol   #1

2:    pragma solidity >=0.6.0 <0.8.0;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L2

[15] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

This change saves 6 gas per instance

There are 6 instances of this issue:

```
File: contracts/RubiconMarket.sol

400:          require(pay_amt > 0);

402:          require(buy_amt > 0);

985:          require(id > 0);

1002:         require(id > 0);
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMa rket.sol#L400

```
File: contracts/rubiconPools/BathHouse.sol

111:            require(_reserveRatio > 0);

281:            require(rr > 0);
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L111

[16] It costs more gas to initialize variables to zero than to let the default of zero be applied

There are 13 instances of this issue:

```
File: contracts/RubiconMarket.sol

990:            uint256 old_top = 0;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L990

```
File: contracts/rubiconPools/BathPair.sol

311:            for (uint256 index = 0; index < array.length; inde

427:            for (uint256 index = 0; index < quantity; index++)

480:            for (uint256 index = 0; index < quantity; index++)

582:            for (uint256 index = 0; index < ids.length; index+
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPoo ls/BathPair.sol#L311

```
File: contracts/rubiconPools/BathToken.sol

635:                 for (uint256 index = 0; index < bonusTokens.le
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L635

```
File: contracts/RubiconRouter.sol

82:             uint256 lastBid = 0;

83:             uint256 lastAsk = 0;

85:             for (uint256 index = 0; index < topNOrders; index+

168:            uint256 currentAmount = 0;

169:            for (uint256 i = 0; i < route.length - 1; i++) {

226:            uint256 currentAmount = 0;

227:            for (uint256 i = 0; i < route.length - 1; i++) {
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L82

[17] `internal` functions not called by the contract should be removed to save deployment gas

If the functions are required by an interface, the contract should inherit from that interface and use the `override` keyword

There are 7 instances of this issue:

```
File: contracts/RubiconMarket.sol

61:       function max(uint256 x, uint256 y) internal pure retur

65:       function imin(int256 x, int256 y) internal pure return

69:       function imax(int256 x, int256 y) internal pure return

76:       function wmul(uint256 x, uint256 y) internal pure retu

441:      function getFeeBPS() internal view returns (uint256) {

959:      function _buys(uint256 id, uint256 amount) internal re

1113      function _offeru(
1114          uint256 pay_amt, //maker (ask) sell how much
1115          ERC20 pay_gem, //maker (ask) sell which token
1116          uint256 buy_amt, //maker (ask) buy how much
1117          ERC20 buy_gem //maker (ask) buy which token
1118:     ) internal returns (uint256 id) {
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L61

[18] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves 6 gas per loop

There are 8 instances of this issue:

```
File: contracts/rubiconPools/BathPair.sol

311:          for (uint256 index = 0; index < array.length; inde

427:          for (uint256 index = 0; index < quantity; index++)

480:          for (uint256 index = 0; index < quantity; index++)

582:          for (uint256 index = 0; index < ids.length; index+
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L311

```
File: contracts/rubiconPools/BathToken.sol

635:                 for (uint256 index = 0; index < bonusTokens.le
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L635

```
File: contracts/RubiconRouter.sol

85:             for (uint256 index = 0; index < topNOrders; index+

169:            for (uint256 i = 0; i < route.length - 1; i++) {

227:            for (uint256 i = 0; i < route.length - 1; i++) {
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L85

[19] Splitting `require()` statements that use `&&` saves gas

See this issue which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper

There are 8 instances of this issue:

```
File: contracts/RubiconMarket.sol

715             require(
716                 !isActive(id) &&
717                     _rank[id].delb != 0 &&
718                     _rank[id].delb < block.number - 10
719:            );
```

```
1177            require(
1178                _rank[id].delb == 0 && //assert id is in the s
1179                    isOfferSorted(id)
1180:           );
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L715-L719

```
File: contracts/rubiconPools/BathPair.sol

120             require(
121                 IBathHouse(_bathHouse).getMarket() !=
122                     address(0x00000000000000000000000000000000
123                     IBathHouse(_bathHouse).initialized(),
124                 "BathHouse not initialized"
125:            );

346             require(
347                 bathAssetAddress != address(0) && bathQuoteAdd
348                 "tokenToBathToken error"
349:            );

419             require(
420                 askNumerators.length == askDenominators.length
421                     askDenominators.length == bidNumerators.le
422                     bidNumerators.length == bidDenominators.le
423                 "not all order lengths match"
424:            );

471             require(
472                 askNumerators.length == askDenominators.length
473                     askDenominators.length == bidNumerators.le
474                     bidNumerators.length == bidDenominators.le
475                     ids.length == askNumerators.length,
476                 "not all input lengths match"
477:            );

506             require(
507                 _bathAssetAddress != address(0) && _bathQuoteA
508                 "tokenToBathToken error"
509:            );
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L120-L125

```
File: contracts/rubiconPools/BathToken.sol

740             require(
741                 recoveredAddress != address(0) && recoveredAdd
742                 "bathToken: INVALID_SIGNATURE"
743:            );
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L740-L743

[20] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

There are 37 instances of this issue:

See original submission for details.

[21] Using `private` rather than `public` for constants, saves gas

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table

There is 1 instance of this issue:

```
File: contracts/rubiconPools/BathToken.sol   #1

70        bytes32 public constant PERMIT_TYPEHASH =
71:           0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L70-L71

[22] Don’t compare boolean expressions to boolean literals

`if (<x> == true)` **=>** `if (<x>)`, `if (<x> == false)` **=>** `if (!<x>)`

There are 4 instances of this issue:

```
File: contracts/rubiconPools/BathHouse.sol   #1

242:                IBathPair(_bathPairAddress).initialized() != t
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L242

```
File: contracts/rubiconPools/BathHouse.sol   #2

372:                approvedStrategists[wouldBeStrategist] == true
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L372

```
File: contracts/rubiconPools/BathPair.sol   #3

149                 IBathHouse(bathHouse).isApprovedStrategist(tar
150:                    true,
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L149-L150

```
File: contracts/rubiconPools/BathToken.sol   #4

228:                IBathHouse(bathHouse).isApprovedPair(msg.sende
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L228

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

Saves deployment costs

There are 12 instances of this issue:

```
File: contracts/RubiconMarket.sol

216:            require(isActive(id));

595:            require(cancel(uint256(id)));

591:            require(buy(uint256(id), maxTakeAmount));

460:            require(!isClosed());

645:            require(!locked, "Reentrancy attempt");

1119:           require(_dust[address(pay_gem)] <= pay_amt);

1209:           require(!isOfferSorted(id)); //make sure offer id

879:                require(offerId != 0);

1002:           require(id > 0);
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L216

```
File: contracts/rubiconPools/BathToken.sol

547            require(
548                owner == msg.sender,
549                "This implementation does not support non-send
550:           );
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathToken.sol#L547-L550

```
File: contracts/RubiconRouter.sol

247:           require(currentAmount >= buy_amt_min, "didnt clear

481:           require(target == ERC20(wethAddress), "target pool
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L247

[24] Division by two should use bit shifting

`<x> / 2` is the same as `<x> >> 1`. The `DIV` opcode costs **5 gas**, whereas `SHR` only costs **3 gas**

There are 4 instances of this issue:

```
File: contracts/RubiconMarket.sol   #1

77:            z = add(mul(x, y), WAD / 2) / WAD;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L77

```
File: contracts/RubiconMarket.sol   #2

81:            z = add(mul(x, y), RAY / 2) / RAY;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L81

```
File: contracts/RubiconMarket.sol   #3

85:            z = add(mul(x, WAD), y / 2) / y;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L85

```
File: contracts/RubiconMarket.sol   #4

89:            z = add(mul(x, RAY), y / 2) / y;
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L89

[25] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables

There are 4 instances of this issue:

```
File: contracts/RubiconMarket.sol   #1
```

```
283:            require(uint128(quantity) == quantity, "quantity i
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L283

```
File: contracts/RubiconMarket.sol   #2

646:            require(_dust[address(pay_gem)] <= pay_amt);
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L646

```
File: contracts/rubiconPools/BathHouse.sol   #3

110:            require(_reserveRatio <= 100);
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L110

```
File: contracts/rubiconPools/BathHouse.sol   #4

111:            require(_reserveRatio > 0);
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L111

[26] Empty blocks should be removed or emit something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be `abstract` and the function signatures be added without any default implementation. If the block is an empty if-statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified (`if(x){}else if(y){...}else{...} => if(!x){if(y){...}else{...}}`)

There are 3 instances of this issue:

```
File: contracts/peripheral_contracts/BathBuddy.sol   #1

69:       receive() external payable {}
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/peripheral_contracts/BathBuddy.sol#L69

```
File: contracts/RubiconRouter.sol   #2

37:       receive() external payable {}
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L37

```
File: contracts/RubiconRouter.sol   #3

39:       fallback() external payable {}
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconRouter.sol#L39

[27] Superfluous event fields

`block.timestamp` and `block.number` are added to event information by default so adding them manually wastes gas

There are 8 instances of this issue:

```
File: contracts/RubiconMarket.sol

131:            uint64 timestamp

142:            uint64 timestamp

154:            uint64 timestamp

165:            uint64 timestamp

177:            uint64 timestamp
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/RubiconMarket.sol#L131

```
File: contracts/rubiconPools/BathHouse.sol

69:             uint256 timestamp,
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathHouse.sol#L69

```
File: contracts/rubiconPools/BathPair.sol

88:             uint256 timestamp,

108:            uint256 timestamp
```

https://github.com/code-423n4/2022-05-rubicon/blob/8c312a63a91193c6a192a9aab44ff980fbfd7741/contracts/rubiconPools/BathPair.sol#L88

[28] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost.

There are 35 instances of this issue:

See original submission for details.

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