Nabla Security Review

Pashov Audit Group

Conducted by: btk, Shaka, Dan Ogurtsov, ZanyBonzy

July 18th 2024 - July 24th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **NablaFinance/contracts-audit-07-2024** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Nabla

Nabla is an AMM that reduces impermanent loss and offers high capital efficiency for trading crypto, real-world, and yielding assets. It uses intelligent pricing with oracles and separates asset provision from risk-taking by having low-risk single-sided Swap Pools and a Backstop Pool that covers the risks, leading to higher returns for liquidity providers and lower swap costs for traders.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 5b44b7bf8bcfa5f33be543884e960e8ddbb202f7

fixes review commit hash - f8c9a257ff3396a739b4ec3a7e1a5b82932e99b3

Scope

The following smart contracts were in scope of the audit:

- `BackstopPoolCore`
- `PythAdapter`
- `OraclePriceAdapter`
- `GenericPool`
- `NablaBackstopPool`
- `NablaPortal`
- `NablaRouter`
- `RouterCore`
- `SwapPool`

7. Executive Summary

Over the course of the security review, btk, Shaka, Dan Ogurtsov, ZanyBonzy engaged with Nabla to review Nabla. In this period of time a total of **16** issues were uncovered.

Protocol Summary

Protocol Name	Nabla
Repository	https://github.com/NablaFinance/contracts-audit-07-2024
Date	July 18th 2024 - July 24th 2024
Protocol Type	DEX

Findings Count

Severity	**Amount**
High	1
Medium	10
Low	5
Total Findings	**16**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Cool-off period for deposits in swap pools can be bypassed	High	Resolved
[M-01]	Pyth oracle price is not validated properly	Medium	Resolved
[M-02]	Function can be called by anyone	Medium	Resolved
[M-03]	Inability to remove swap pools from the backstop pool	Medium	Acknowledged
[M-04]	Excess ETH is not refunded	Medium	Acknowledged
[M-05]	Insurance timelock is flawed	Medium	Resolved
[M-06]	Fee-on-transfer and rebase tokens.	Medium	Acknowledged
[M-07]	swapIntoFromRouter missing the checkPoolCap modifier	Medium	Resolved
[M-08]	Users can omit to update the price feed to use stale prices	Medium	Acknowledged
[M-09]	SwapPool's are vulnerable to flashloan attacks	Medium	Acknowledged
[M-10]	Arbitrage opportunity using different prices in the same block	Medium	Acknowledged
[L-01]	Certain swap pools cannot be unregistered	Low	Acknowledged
[L-02]	No function to unregister assets	Low	Resolved
[L-03]	Non-allowed users can withdraw shares	Low	Acknowledged
[L-04]	Fees can be bypassed by redeeming in small fragments	Low	Acknowledged

[L-05]	BackstopPool is vulnerable to inflation attack	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Cool-off period for deposits in swap pools can be bypassed

Severity

Impact: Medium

Likelihood: High

Description

Deposits into the swap pool are subject to a cool-off period before being available for withdrawal via the backstop pool.

However, this restriction can be bypassed by transferring the LP tokens received from the deposit to another address.

Proof of concept

Add the following code to the file `BackstopPool.t.sol` and run `forge test --mt test_bypassDepositCoolOff`.

```
function test_bypassDepositCoolOff() public
    changeSwapPoolCoverageTo(swapPool2, 0.2e18)
{
    uint256 initialUsdBalance = usd.balanceOf(address(this));
    uint256 initialAsset2Balance = asset2.balanceOf(address(this));

    // Setup
    vm.roll(20e6);
    address secondaryAccount = makeAddr("secondaryAccount");
    deal(secondaryAccount, 100);
    backstop.setInsuranceFee(address(swapPool2), 100);
    bytes[] memory _mockUpdateData = _makeMockPriceUpdateData(2);

    // Deposit 10 asset2 into swapPool2 and transfer LP tokens to secondary
    // account
    (uint256 lpTokens, ) = swapPool2.deposit(10e18, 0, block.timestamp);
    swapPool2.transfer(secondaryAccount, lpTokens);

    // From secondary account, redeem LP tokens and transfer USD token to main
    // account
    vm.startPrank(secondaryAccount);
    asset2.approve(address(swapPool2), MAX_UINT256);
    backstop.redeemSwapPoolShares{value: 2}(
        address(swapPool2),
        lpTokens,
        0,
        block.timestamp,
        _mockUpdateData
    );
    usd.transfer(address(this), usd.balanceOf(secondaryAccount));
    vm.stopPrank();

    // The result is an instant profit > 9%
    uint256 usdReceived = usd.balanceOf(address(this)) - initialUsdBalance;
    uint256 asset2Spent = initialAsset2Balance - asset2.balanceOf(address
      (this));
    uint256 asset2SpentInUsd = asset2Spent * ASSET2_PRICE / 1e18;
    uint256 profit = usdReceived - asset2SpentInUsd;
    assert(profit > 0.9e18);
}
```

Recommendations

A solution can pass for overriding the `_transfer` function in the `SwapPool` contract to update the `latestDepositAtBlockNo` value for the recipient address. However, this could be used for DoS withdrawals via the backstop pool by sending 1 wei to the recipient address. A better solution would be storing and updating the number of tokens in cool-off for each address, along with the block number of the last deposit or transfer received. This way, users would be able to withdraw the amount of tokens that exceed the tokens in the cool-off state.

8.2. Medium Findings

[M-01] Pyth oracle price is not validated properly

Severity

Impact: High

Likelihood: Low

Description

`PythAdapter.getAssetPrice` does not perform input validation on the `price`, `conf`, and `expo` values, which can lead to the contract accepting invalid or untrusted prices.

It is especially important to validate the confidence interval, as stated in the Pyth documentation, to prevent the contract from accepting untrusted prices.

Recommendations

```
-       (int64 price, int32 expo) = (
+       (int64 price, uint64 conf, int32 expo) = (
            pythStructsPrice.price,
+           pythStructsPrice.conf,
            pythStructsPrice.expo
        );

+       if (price <= 0 || expo < -18) {
+           revert("PA:getAssetPrice:INVALID_PRICE");
+       }
+
+       if (conf > 0 && (price / int64(conf) < MIN_CONFIDENCE_RATIO)) {
+           revert("PA:getAssetPrice:UNTRUSTED_PRICE");
+       }
```

[M-02] Function can be called by anyone

Severity

Impact: Low

Likelihood: High

Description

In the `NablaRouter` contract, the `swapExactTokensForTokens` function uses the `allowed` modifier to check if the caller is allowed to access the function. However, the `swapExactTokensForTokensWithoutPriceFeedUpdate` function does not have this modifier.

As so, any address can bypass the access check in `swapExactTokensForTokens` by calling `PriceOracleAdapter.updatePriceFeeds` and `swapExactTokensForTokensWithoutPriceFeedUpdate`.

Recommendations

Add the `allowed` modifier to `swapExactTokensForTokensWithoutPriceFeedUpdate`.

[M-03] Inability to remove swap pools from the backstop pool

Severity

Impact: High

Likelihood: Low

Description

In the backstop pool, once a swap pool is added to the list of covered pools, it cannot be removed.

If the Pyth network deprecated the price feed for the asset of one of the covered pools, or it began returning bad data, the execution of `_getAllTokenPrices` would revert, making it impossible to deposit or withdraw from the backstop pool and, consequently, locking the funds of the LPs in the covered pool.

Recommendations

Give to the owner of the backstop pool the ability to remove swap pools from the list of covered swap pools.

[M-04] Excess ETH is not refunded

Severity

Impact: High

Likelihood: Low

Description

In multiple instances, pricefeeds are updated without the excess eth sent being refunded. This will result in the funds being stuck in the contracts forever.

1. NablaRouter.sol - `swapExactTokensForTokens`
2. NablaBackstopPool.sol - `deposit`, `finalizeWithdrawBackstopLiquidity`, `redeemSwapPoolShares`, `finalizeWithdrawExcessSwapLiquidity`, `redeemCrossSwapPoolShares`
3. NablaPortal.sol - `swapExactTokensForEth`, `_updatePriceFeeds`

Recommendations

Two potential fixes:

1. Implement a check for update fees in these functions, compare that msg.value sent is enough, and refund any excess tokens.

A roughly similar logic can be found in `swapEthForExactTokens` function, or the below code can be adapted.

```
//...
        uint256 updateFee = getUpdateFee
        //(oracleAdapter, _priceUpdateData); //@note a function to query fee can be cr
        uint256 refundAmount = msg.value - updateFee;
        if (refundAmount > 0) {
            (bool success, ) = msg.sender.call{value: refundAmount}("");
            require(success, "NP:swapEthForExactTokens:REFUND_FAILED");
        }
```

2. Creating a sweep function in these contracts, that admin can use to recover any excess tokens in the contracts.

[M-05] Insurance timelock is flawed

Severity

Impact: Medium

Likelihood: Medium

Description

When a user deposits in the swappool, the latest deposit block number is registered. This is to prevent users from immediately redeeming swappool lp tokens for backstop assets through the `backstopBurn` function without first waiting for the needed insurance period. However, the way this is tracked, involves a direct mapping of the block.number to the sender, as such, every new deposit a user makes overwrites the previously tracked `block.number`. As a result, serious pool depositors will be forced to wait longer for time periods before being able to redeem their lptokens.

```
function deposit(
//...
        latestDepositAtBlockNo[msg.sender] = block.number;
//...
        _processDeposit(_depositAmount, sharesToMint_);
//...
    }
```

The same can be observed in the `initiateWithdraw` function in BackstopPoolCore.sol, and appears to be by design based on the code comment on the function.

Recommendations

Recommend tracking each deposit with an id, and its corresponding `block.number` instead. That way, users, especially constant pool depositors will not be forced to endure longer waiting periods.

[M-06] Fee-on-transfer and rebase tokens.

Severity

Impact: High

Likelihood: Low

Description

In various parts of the codebase, assets are transferred using with the assumption that the amount transferred, is the amount received. This however isn't the case when dealing with certain tokens. i. Some charge a fee during transfers, e.g PAXG. Also important to note is that tokens like USDT also have the option to charge fees, but currently do not. ii. Some tokens, notably stETH have a 1 to 2 wei corner case, in which the amount received during a transfer is less than the amount specified. iii. Some tokens rebase, both positively and negatively, in which the holder's balance overtime increases or decreases. stETH also does this. This also includes tokens that give airdrops and the likes.

These tokens have the ability to mess with the protocol's accounting if in use. This is because the transfer functions aren't optimized to handle them, and as a result can lead to situations in which the pools' asset balance will be way less than the amount expected and tracked. Here, the protocol will incur extra costs to cover for these situations. Also, the tokens received from airdrops and positive rebases can be lost forever as there's no way to retrieve them.

The following are the functions affected.

1. NablaPortal.sol - `swapExactTokensForEth` #L234, `swapExactTokensForTokens` #L293
2. GenericPool.sol - `_processDeposit` #L96, `_processWithdrawal` #L118
3. BackstopPoolCore.sol - `_redeemSwapPoolShares` L466,
4. SawPool.sol - `backstopDrain` #L511, `swapIntoFromRouter` #L585, `swapOutFromRouter` #L670 #L675,
5. RouterCore.sol - `_executeSwap` #L230 #L245,

Recommendations

Before any token transfer, both in and out of the protocol, recommend checking the contract balance before and after, and registering the difference as

the amount sent. This helps handle fee-on-transfer tokens, and the 1 wei corner cases. For the rebasing tokens and variable balances, a system of balance tracking and excess token sweep functions can be implemented to periodically skim the excess tokens from the contrascts to prevent them from being lost.

Alternativly, explicitely blocklisting these token types to prevent them from being made pool assets.

[M-07] `swapIntoFromRouter` missing the `checkPoolCap` modifier

Severity

Impact: Medium

Likelihood: Medium

Description

When depositing into the pools, the pool cap is enforced. This is done through the `checkPoolCap` modifier. The modifier tracks the pool's asset's balance, with the amount being deposited, and compares them against the pool cap.

```
modifier checkPoolCap(uint256 _additionalDeposit) {
        require(poolAsset.balanceOf(address
          (this)) + _additionalDeposit <= poolCap, "deposit: CAP_EXCEEDED");
        _;
    }
```

This means that any tokens entering the pool potentially add up to the cap, regardless of whether it's through the `deposit` function or not.

This also occurs in the `swapIntoFromRouter` function, as the pool asset is transferred from the router to the pool, raising the pool's asset balance, and indirectly raising the pool cap requirement for the pool being swapped into. As a result, the pool cap can easily be exhausted and even bypassed through swaps. When this occurs, users will not be able to deposit into the pool.

```
function swapIntoFromRouter(
        uint256 _amount
    )
        external
        nonReentrant
        onlyRouter
        whenNotPaused
        returns (uint256 effectiveAmount_)
    {
        effectiveAmount_ = _quoteSwapInto(_amount);
        reserve = reserve + effectiveAmount_;
        reserveWithSlippage = reserveWithSlippage + _amount;

        poolAsset.safeTransferFrom(msg.sender, address(this), _amount);
    }
```

Other important factors to note with the pool cap:

1. It can be intentionally or unintentionally dossed by whales;
2. A potential race condition when pool cap is about to be reached;
3. Malicious users can send large amounts of the asset to the pool, indirectly raising the pool balance. This works best if the tokens' worth is not very much making the attack cheaper, or in a coordinated effort by various attackers.

Recommendations

Consider adding the `checkPoolCap` modifier to the `swapIntoFromRouter`, or tracking the pool assets by source. And only querying assets from deposits in the `checkPoolCap` modifier.

[M-08] Users can omit to update the price feed to use stale prices

Severity

Impact: High

Likelihood: Low

Description

[!NOTE]
All functions that internally call `_updatePriceFeeds()` are

vulnerable to this attack.

When invoking any function in the Nabla protocol, users must provide an update fee and `_priceUpdateData()` to update the asset prices they intend to use. For example, consider the `redeemCrossSwapPoolShares()` function:

```
function redeemCrossSwapPoolShares(
        address _swapPool,
        address _targetSwapPool,
        uint256 _shares,
        uint256 _minAmount,
        uint256 _deadline,
        bytes[] calldata _priceUpdateData
    )
        external
        payable
        nonReentrant
        whenNotPaused
        returns (uint256 finalAmount_)
    {
        _updatePriceFeeds(address(router.oracleAdapter()), _priceUpdateData);

        (finalAmount_, ) = _executeRedeemCrossSwapPoolShares(
            _swapPool,
            _targetSwapPool,
            _shares,
            _minAmount
        );
    }
```

As shown above, `_updatePriceFeeds()` is called with the user-supplied `_priceUpdateData` array. The protocol assumes that users will always provide valid price feed data, but malicious users can exploit this assumption. They can pass irrelevant token data to `_updatePriceFeeds()`, effectively using stale prices as long as they are within the `priceMaxAge` limit.

For instance, if a swap pool asset is a highly volatile token whose price fluctuates within a minute, an attacker could exploit this by calling `redeemCrossSwapPoolShares()` without updating the price feeds, thus receiving the token at a discount.

Recommendations

Implement a standard function to compute a `_priceUpdateData` array for a specific asset to ensure that price feeds are consistently updated.

[M-09] SwapPool's are vulnerable to flashloan attacks

Severity

Impact: High

Likelihood: Low

Description

SwapPool allows users to provide liquidity (LP) and earn fees. However, the ability for liquidity providers to deposit and withdraw within the same block makes the system vulnerable to flash loan attacks, which can exploit other LPs' rewards. A malicious user can front-run a legitimate deposit with a flash loan deposit and then back-run it with a withdrawal in the same block, effectively siphoning a portion of the LP fees. Below is a proof-of-concept demonstrating the attack:

```
function testFlahloan()
        public
        changeSwapPoolCoverageTo(pool, 0.3e18)
    {
        uint256 _accumulatedSlippageInitial = pool.reserveWithSlippage() -
            pool.reserve();

        // attacker frontrun the user deposit with 10m flashloan
        address attacker = makeAddr("attacker");
        uint256 flashloan = 10_000_000 ether;

        vm.startPrank(attacker);
        asset.approve(address(pool), MAX_UINT256);
        asset.mint(attacker, flashloan);
        (uint256 _lpTokens,) = pool.deposit
          (flashloan, 0, block.timestamp + USER_DEADLINE_TOLERANCE);
        vm.stopPrank();

        // user deposit 100k assets to the pool
        pool.deposit(
            100_000 ether,
            0,
            block.timestamp + USER_DEADLINE_TOLERANCE
        );

        // the attacker withdraw his shares and rip 0.01193665464548911 assets
        vm.prank(attacker);
        pool.withdraw(
            _lpTokens,
            flashloan,
            block.timestamp + USER_DEADLINE_TOLERANCE
        );
    }
```

In this example, the attacker earns 0.01193665464548911 assets (e.g., $11 if the asset price is $1000). The lower the `_targetCoverageRatio` (set to 0.3e18 in this PoC), the greater the attacker's gains.

Recommendations

To mitigate this vulnerability, consider enforcing a delay between deposits and withdrawals.

[M-10] Arbitrage opportunity using different prices in the same block

Severity

Impact: Medium

Likelihood: Medium

Description

The current implementation of the `PythAdapter` contract allows using different prices for the same token in the same block. This can be exploited by a malicious user to make a profit with zero risk.

There are different ways of performing an operation without updating the price feed of a certain token:

- Omit the target token in the array sent to `updatePriceFeeds`.
- Submit an outdated price for the target token (`updatePriceFeeds` does not revert in this case).
- Swap through `swapExactTokensForTokensWithoutPriceFeedUpdate`.

This way, a user can perform a swap with an outdated price, and then perform an operation in the opposite direction with the updated price.

> Note that even if it was enforced to successfully update the price feed before any operation, it is still possible to submit later a more recent price in another operation performed in the same block, making it still possible to arbitrage.

Proof of concept

At timestamp `t` the prices in the Pyth network are as follows:

- WBTC/USD: $50,000
- USDC/USD: $1

At block `b` `updatePriceFeeds` is called with the above prices.

At timestamp `t + 10` the `WBTC/USD` price increases to $51,000.

At block `b + 1` Alice calls `swapExactTokensForTokens` without updating the `WBTC` price, and swaps 50,000 USDC for 1 WBTC. In the same block, Alice calls `swapExactTokensForTokens` again, this time updating the `WBTC` price to $51,000, and swaps 1 WBTC for 51,000 USDC, making a profit with zero risk.

In this example, we have not taken into account the fees, for simplicity. The trade will be profitable as long as `total fees < Δ price * amount`. In this example, if the fees are less than $1,000 (2%), Alice would have made a profit.

Recommendations

A possible solution would be to store the current price of a token the first time it is used in a block, and then use this price for the rest of the block. This way, the price of a token would be consistent throughout the block.

8.3. Low Findings

[L-01] Certain swap pools cannot be unregistered

In RouterCore.sol, the owner can call the `unregisterPool` function to unregister pools. This is then followed by an approval to 0 to remove the pool's prior approval to the asset. Some tokens e.g BNB revert on zero value approval. As a result, such pools cannot be unregistered.

```
function unregisterPool(
        address _asset
    ) external onlyOwner returns (bool success_) {
        require(_asset != address(0), "RC:unregisterPool:NO_ASSET");
        address pool = address(poolByAsset[_asset]);
        require(pool != address(0), "RC:unregisterPool:NO_POOL");

        delete poolByAsset[_asset];
        IERC20(_asset).safeApprove(pool, 0);

        emit SwapPoolUnregistered(msg.sender, _asset);
        return true;
    }
```

The best fix for this is to watch out for those types of tokens and not register them in the first place. To fix this in code, the `safeDecreaseAllowance` function can be used instead, to decrease the pool's allowance from `type(uint256).max` to a very negligible amount, e.g. 1 wei. The setup can be wrapped in a try-catch block.

[L-02] No function to unregister assets

NablaPortal.sol has the `registerAsset` function to link an asset to a router, but no function to unlink it. If an asset or the router gets compromised, there's no way for such assets to be unregistered. Recommend introducing a permissioned function to do just that, the function should also clear the router's allowance. See the implementation of `unregisterPool` in RouterCore.sol.

[L-03] Non-allowed users can withdraw shares

In an NFT-gated swap pool, only authorized users can deposit into the pool:

```
function deposit(
        uint256 _depositAmount,
        uint256 _minLPAmountOut,
        uint256 _deadline
    )
        external
        nonReentrant
        whenNotPaused
        allowed
        returns (uint256 sharesToMint_, int256 fee_)
    {
```

However, non-authorized users can withdraw shares. Users can purchase shares directly from the open market and withdraw them from the pool, even if they do not have deposit access. To mitigate this, consider adding an allowed modifier to the withdraw function.

[L-04] Fees can be bypassed by redeeming in small fragments

The NablaBackstopPool contract supports all ERC20 tokens. When users redeem swap pool shares, the pool takes an insurance fee calculated as follows:

```
uint256 fee = (backstopTokenAmount * swapPoolConfig.insuranceFeeBps) /
            INSURANCE_FEE_PRECISION;

        unchecked {
            amountOut_ = backstopTokenAmount - fee;
        }
```

Users can exploit this calculation by making small, fragmented redemptions, causing the fee to round down to zero. This issue is particularly problematic for tokens with low decimal precision, such as GUSD, which has 2 decimals. Consider the following scenario:

> [!IMPORTANT]
> The attack is only effective on L2s with low gas fees.

- `poolAsset` = GUSD
- `insuranceFeeBps` = 100
  - Bob calls `redeemSwapPoolShares()` with `_shares = 99` assuming 1:1 conversion between shares to assets.
  - Calculation:
    - `(backstopTokenAmount * insuranceFeeBps) / INSURANCE_FEE_PRECISION = (99 * 100) / 10000 = 0`
    - The insurance fee rounds down to zero, meaning the pool receives no fees for each redemption of 99 tokens.

To prevent this exploit, consider adding the following check:

```
if (swapPoolConfig.insuranceFeeBps > 0) {
        require(
            (backstopTokenAmount * swapPoolConfig.insuranceFeeBps) >=
                INSURANCE_FEE_PRECISION,
            "BC:_redeemSwapPoolShares:ZERO_FEES"
        );
}
```

[L-05] BackstopPool is vulnerable to inflation attack

- The BackstopPool, which operates similarly to an ERC4626 vault, is susceptible to an inflation attack.
- An attacker can exploit this vulnerability by making a minimal initial deposit (e.g., 1 wei), and then artificially inflating the totalPoolWorth.
- This manipulation causes the base share price to be significantly higher, affecting all subsequent deposits.
- Due to rounding down, if the malicious initial deposit front-runs other deposits, the attacker will receive 1 wei worth of shares at the inflated price.

Below is a proof-of-concept (PoC) demonstrating this attack:

```
function testInflationAttack() public {
        bytes[] memory _mockUpdateData = _makeMockPriceUpdateData
          (BACKSTOP_ASSETS_QUANTITY);
        uint256 _updateFee = pythAdapter.getUpdateFee(_mockUpdateData);
        address attacker = makeAddr("Attacker");
        address victim = makeAddr("Victim");

        setupPostion(attacker);
        backstop.deposit{value: _updateFee}(
            1 wei,
            0,
            block.timestamp + USER_DEADLINE_TOLERANCE,
            _mockUpdateData
        );
        usd.transfer(address(backstop), 100 ether);
        vm.stopPrank();

        setupPostion(victim);
        backstop.deposit{value: _updateFee}(
            200 ether,
            0,
            block.timestamp + USER_DEADLINE_TOLERANCE,
            _mockUpdateData
        );
        vm.stopPrank();

        vm.prank(attacker);
        backstop.initiateWithdraw(1 wei);

        vm.warp(INITIAL_WITHDRAWAL_DELAY_PERIOD + 1);

        uint256 balanceBefore = IERC20(backstop.asset()).balanceOf(attacker);

        vm.prank(attacker);
        backstop.finalizeWithdrawBackstopLiquidity{value: _updateFee}(
            1 wei,
            block.timestamp + USER_DEADLINE_TOLERANCE,
            _mockUpdateData
        );

        uint256 balanceAfter = IERC20(backstop.asset()).balanceOf(attacker);

        console.log(balanceAfter - balanceBefore); // 150e18
    }
```

As demonstrated, the attacker successfully gained 50e18 assets.

Full setup: link

Consider minting a fixed amount of shares for the first deposit or use "virtual shares" as it is implemented in ERC4626.