decurity

Smart Contract Security Audit Report

EtherFi

1. Contents

2. General Information

This report contains information about the results of the security audit of the EtherFi (hereafter referred to as "Customer") smart contracts, conducted by Decurity in the period from 04/01/2024 to 04/08/2024.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following PRs:

- Liquifier integration with EigenLayer m2 and PancakeSwap (+ other chores): https://github.com/etherfi-protocol/smart-contracts/pull/28 (initial commit 32424a24a96ba90db4f80f0796392c55db7a9d29, retest commit 1e4672fd8eafecb2e64b6d928446d54c843ef879),
- Liquifier integration with the SyncPools (LayerZero bridging): https://github.com/etherfi-protocol/smart-contracts/pull/30 (commit 674cae183d39812979c78eda063aa135c7c5813b).

The audit only focused on the actual code diff. The rest of the repository was out of scope, however, we did best effort to study the previous audit reports and the overall workflow to make sure the integrations are implemented correctly.

2.3. Threat Model

The assessment presumes the actions of an intruder who might have the capabilities of any role (an external user, token owner, token service owner, or a contract). The risks of centralization were not taken into account at the Customer's request.

The main possible threat actors are:

- User,
- Protocol owner,
- Protocol admins,
- 3rd party (LST, EL, LayerZero) owners/contracts.

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, and the impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises the best effort to perform its contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using limited resources.

3. Summary

As a result of this work, we have discovered no critical exploitable security issues.

The other suggestions included fixing the low-risk issues and some best practices (see Security Process Improvement).

The EtherFi team has given feedback for the suggested changes and an explanation for the underlying code.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of April 8, 2024.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
[PR-28] Peg to `stETH` leads to arbitrage opportunities	src/Liquifier.sol	Low	Acknowledged
[PR-28] The admins can sweep tokens from `eigenPod`	src/EtherFiNodesManager.sol	Low	Fixed
[PR-28] The admin can sweep the tokens using `pancakeSwapForEth`	src/Liquifier.sol	Low	Acknowledged
[PR-30] Outdated comment	src/Liquifier.sol	Info	Not fixed
[PR-28] Front-running of `claimWithdraw` possible	src/WithdrawRequestNFT.sol	Info	Acknowledged
[PR-28] Broken back compatibility for `cbETH` and `wbETH`	src/Liquifier.sol	Info	Fixed

4. General Recommendations

This section contains general recommendations on how to improve the overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. [PR-28] Peg to `stETH` leads to arbitrage opportunities

Risk Level: Low

Status: Acknowledged

Contracts:

- src/Liquifier.sol

Location: Lines: 377.

Description:

The `quoteByMarketValue` function implies that `1 ETH = 1 stETH` and allows minting `eETH` in 1:1 correspondence with `stETH`. However, there're discrepancies in `stETH`'s price sufficient for the arbitrage.

The following test case demonstrates the arbitrage opportunity (0.4 ETH profit on the 30 ETH deposit at the moment of writing) caused by `stETH`'s price being lower than `ETH`:

```
interface ICurvePool_payable {
    function exchange(int128 i, int128 j, uint256 dx, uint256 min_dy) external
payable returns (uint256);
}

// . . .

    function test_arbitrage_steth() public {
        _setUp(MAINNET_FORK);

        // someone deposits directly to the liquidity pool
        liquidityPoolInstance.deposit{value: 1000 ether}(address(0));

        uint256 amount = 30 ether;

        // now Alice deposits to the liquifier
        vm.deal(alice, amount);
        uint256 before = alice.balance;

        vm.startPrank(alice);

        ICurvePool_payable pool =
ICurvePool_payable(0xDC24316b9AE028F1497c275EB9192a3Ea0f67022);
        pool.exchange{value:amount}(0, 1, amount, 0);
        amount = stEth.balanceOf(alice);

        stEth.approve(address(liquifierInstance), amount);

        liquifierInstance.depositWithERC20(address(stEth), amount,
address(0));

        // now Alice wants to withdraw from the liquidity pool
        eETHInstance.approve(address(liquidityPoolInstance), amount);
        uint256 id_alice = liquidityPoolInstance.requestWithdraw(alice,
amount);
        vm.stopPrank();

        _finalizeWithdrawalRequest(id_alice);

        vm.prank(alice);
        withdrawRequestNFTInstance.claimWithdraw(id_alice);

        console.log('balance: %s', alice.balance);
        console.log('before: %s', before);
        console.log('profit: %s', alice.balance - before);
        assertGt(alice.balance, before);
    }
```

Remediation:

Consider using the price oracles resistant to flash loans.

5.2. [PR-28] The admins can sweep tokens from `eigenPod`

Risk Level: Low

Status: Fixed in the commit 22f9bddc.

Contracts:

- src/EtherFiNodesManager.sol

Location: Lines: 418.

Description:

In `EtherFiNode` contract the `callEigenPod` function performs `_verifyEigenPodCall(data)` check before calling `Address.functionCall(eigenPod, data)`. The function validates the recipient field only when the selector is equal to the `withdrawNonBeaconChainETHBalanceWei(address recipient, uint256 amountToWithdraw)` function.

However, the function does not validate the recipient address when calling `eigenPod.recoverTokens()`. The `EigenLayerOperatingAdmin` user can transfer ERC20 tokens from the eigenPod contract to any address.

Remediation:

Add a similar check for the `recoverTokens` function selector.

5.3. [PR-28] The admin can sweep the tokens using `pancakeSwapForEth`

Risk Level: Low

Status: Acknowledged

Contracts:

- src/Liquifier.sol

Location: Lines: `301`.

Description:

In the `Liquifier` contract the new `pancakeSwapForEth` function allows a user with the admin role to call `pancakeRouter.exactInputSingle` with arbitrary parameters to exchange tokens from the `Liquifier` contract to ETH.

The caller can specify `amountOutMinimum = 0` effectively making slippage equal to 100% for this swap. Under these conditions, the admin user can make a sandwich attack on this swap and swap all tokens (for example, `stETH`) to `ETH` at a very unfair rate. In other words, a user with the admin role can steal all ERC20 tokens on this contract.

Remediation:

Add mandatory minimum slippage protection to the call.

5.4. [PR-30] Outdated comment

Risk Level: Info

Status: Not fixed

Contracts:

- src/Liquifier.sol

Location: Lines: 263.

Description:

The treasury fees have been removed but the comment before the `withdrawEther` function still states otherwise:

```
    // Send the redeemed ETH back to the liquidity pool & Send the fee to
Treasury
```

Remediation:

Update the comment.

5.5. [PR-28] Front-running of `claimWithdraw` possible

Risk Level: Info

Status: The admins will call the `WithdrawRequestNFT.invalidateRequest` function to actually disallow someone to withdraw.

Contracts:

- src/WithdrawRequestNFT.sol

Description:

The withdrawal request ids in the `WithdrawRequestNFT` contract are subsequent. Therefore, if the admin finalizes a request, all the prior requests are also finalized. If the admin did not intend to allow the Alice's request but finalized the subsequent Bob's request, Alice can front-run the Bob's claim and get the locked liquidity as shown in the following self-explanatory test case:

```
    function test_frontrun_claimwithdraw() public {
        _setUp(MAINNET_FORK);

        // someone deposits directly to the liquidity pool
        liquidityPoolInstance.deposit{value: 10 ether}(address(0));

        // now Alice deposits to the liquifier
        vm.deal(alice, 1000 ether);
        uint256 before = alice.balance;

        vm.startPrank(alice);
        stEth.submit{value: 1 ether}(address(0));
        stEth.approve(address(liquifierInstance), 1 ether);

        liquifierInstance.depositWithERC20(address(stEth), 1 ether,
address(0));

        // now Alice wants to withdraw from the liquidity pool
        // however, her withdrawal request is not finalized
        eETHInstance.approve(address(liquidityPoolInstance), 1 ether);
        uint256 id_alice = liquidityPoolInstance.requestWithdraw(alice, 1
ether);
        vm.stopPrank();

        // now Bob also deposits to the liquifier
        vm.deal(bob, 1000 ether);

        vm.startPrank(bob);
        stEth.submit{value: 10 ether}(address(0));
        stEth.approve(address(liquifierInstance), 10 ether);

        liquifierInstance.depositWithERC20(address(stEth), 10 ether,
address(0));

        // Bob requests withdrawal from the liquidity pool
        eETHInstance.approve(address(liquidityPoolInstance), 10 ether);
        uint256 id_bob = liquidityPoolInstance.requestWithdraw(bob, 10 ether);
        vm.stopPrank();

        // his request is finalized but he does not claim
        _finalizeWithdrawalRequest(id_bob);

        // Alice claims her withdrawal although it was not finalized
        vm.prank(alice);
        withdrawRequestNFTInstance.claimWithdraw(id_alice);

        assertGe(alice.balance, before);

        // now unsuspecting Bob tries to claim his ETH and fails
        vm.prank(bob);
        vm.expectRevert(0xbb55fd27); // InsufficientLiquidity()
        withdrawRequestNFTInstance.claimWithdraw(id_bob);
    }
```

However, this is not a vulnerability because the Alice’s request can be manually invalidated in case if the admin wishes so.

Remediation:

No code fixes needed. Follow the intended invalidation procedure to avoid front-running.

5.6. [PR-28] Broken back compatibility for `cbETH and wbETH`

Risk Level: Info

Status: Fixed in the commit 1e4672fd.

Contracts:

- src/Liquifier.sol

Description:

The `depositWithQueuedWithdrawal` function and the corresponding user flow is deprecated, however, the code is still there with the removed cap check in `_enqueueForWithdrawal` for the sake of backward compatibility with the already queued withdrawals.

The check removal does not allow to bypass the 0 cap for the `wbETH` and `cbETH` tokens as we checked, however, these tokens have also been removed from `quoteByMarketValue` function which would make it impossible to processed the already queued withdrawals because the call to `quoteStrategyShareForDeposit` would fail.

Remediation:

Bring the tokens back to `quoteByMarketValue`.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.