DefiEdge Finance Ethereum Contracts Security Audit

September 30th, 2021 (updated December, 13th, 2021)

@lucash-dev

https://hackerone.com/lucash-dev

https://github.com/lucash-dev

Disclaimer

Security audits are inherently limited -- there is no possibility of strict proofs that a given system doesn't contain further vulnerabilities. The best that can be offered is a good faith effort to find as many vulnerabilities as possible within the resource constraints of the project.

Given the above, this audit report is provided "AS IS", without any guarantee of correctness or completeness. **There is no warranty of any kind**. The author takes no responsibility for any loss or damage caused by the use, misuse or failure to use the information contained in this document, as well as for any information that might be missing from it (e.g. missed vulnerabilities).

By using the information contained in this report, you agree to do so at your own risk, and not to hold the authors liable for any consequences of such use.

Summary

This document contains the results of the audit conducted on the DefiEdge Ethereum smart contracts.

- **Methodology**: a general description of the methodology used to find possible issues.
- **Exploitable Vulnerabilities**: a list of immediately exploitable issues found.
- **Potential Vulnerabilities and Trust issues**: a list of issues that might be exploitable depending on external factors, or by privileged users.
- **Limitations**: description of factors that limit the completeness of the analysis presented in this report.

Scope

The scope of this audit are the Ethereum smart contracts found at

- https://github.com/unbound-finance/defiedge-core/tree/a050d9ad84d18c340964d42a71ba487b0e8ad7e8
- https://github.com/unbound-finance/defiedge-periphery/tree/510e0c056e38e54a247155ad5f8bb87f46865b9e

The contracts were later reviewed at commits 353bc236e82fdfb672208e551b2465f258922336.

The audit is focused on identifying implementation defects that can lead to security vulnerabilities, and does not constitute an appraisal of the code's value proposition.

Methodology

The audit was conducted manually by an experienced security researcher, by inspecting the code in two different ways:

- Systematic review of every function in the contracts, spotting missing best practices and logical flaws.
- Free exploration of interaction points and execution flows, searching for concrete attack vectors and

invalid assumptions.

While the first approach mimics a more traditional code review, the second one tries to reproduce the kind of coverage you would obtain from a "bug bounty" program.

It's the author's belief that combining these approaches offer much higher likelihood of catching high-severity issues than using either individually.

Exploitable Vulnerabilities Found

ShareRewards initialization

One exploitable vulnerability was found in the contract "ShareRewards" though the impact is unclear.

The function `init` in the `ShareRewards` contract has two related issues that allow for unintended use:

1. There is no restriction on the function being called more than once.
2. There is no restriction on which `msg.sender` can call the function.
3. The token contract passed as argument isn't validated in any way.

Looking at the code

```
uint256 balance = dummyToken.balanceOf(msg.sender);
...
MASTER_CHEF.deposit(MASTER_PID, balance);
```

It's clear that by passing a maliciour contract as the `dummyToken` argument, an attacker can make at any time the `ShareRewards` contract call `MASTER_CHEF.deposit` with whatever balance as parameter.

The full impact of this isn't clear as `MASTER_CHEF` isn't in scope for this audit, and it would depend on precise implementation and configuration details.

From comments in the code, however, it might be possible for the attacker to force it to mint a quantity of `TOKEN`.

Recomended Remedy: add the `onlyOwner` modifier to the `init` function.

UPDATE: the development team updated the code, removing the vulnerable function in commit `5c9f8d6fa674ebb2c55008ac939195d224147752`.

Creating strategies with "evil pools" (phishing)

The contract `DefiEdgeStrategyFactory` allows for any user to deploy a new `DefiEdgeStrategy` that uses any given contract as its underlying pool.

All the code *assumes* the pool is in fact a `UniswapV3Pool` but the factory `createStrategy` functions doesn't check for it at any time.

That means that a malicious user can create a malicious pool contract with any implementation even while pointing to legitimate tokens.

In other words, it isn't enough for the user to validate the tokens in the pool are trusted -- each pool contract itself would need to be audited/trusted by users.

That opportunity for phishing could easily be prevented by adding a `require` statement to the factory to check if the pool contract is in fact a valid `UniswapV3Pool`.

As an example impact, a malicious pool contract could transfer all tokens deposited by the strategy to the attacker.

Suggested Remedy: add logic to validate the address of the pool contract as pointing to a valid

`UniswapV3Pool`.

UPDATE: The development team implemented a fix for this issue in commit `5f9dab1c9da95f13a541b0292ce63d5b70bdc68f`, following the suggestion in this report.

Potential Vulnerabilities and Trust issues

Possibility of "Evil Operators"

Strategy operators have considerable control over the funds deposited into the strategy.

This seems to be by design, as the operator will be responsible for rebalancing the strategy and setting parameters (Uniswap ticks) for depositing liquidity.

Furthermore the `swap` method allows for setting arbitrary minimum amounts which could lead to an attack emptying the Strategy as follows:

1. Obtain in flash loan large amount of Token0
2. Buy from pool large amount of Token1, driving price up.
3. Call strategy.swap to swap all the stragey's Token0 for a small amount of Token1, further driving up the price.
4. Sell back amount of Token1 obtained in 2.
5. Payback flashloan from 1.

If liquidity is low in the pool, the operator could easily drain the strategy of most tokens using this procedure.

This scenario can only be performed by the *operator*. It is very important to make it abundantly clear to users that depositing in any strategy means implicitly trusting your funds to the operator -- who can be anyone, not necessarily affiliated with the project.

Changing this to prevent operators from "rug pulling" would require huge modifications to design though, but mitigating with clear communication is still important.

Suggested Mitigation: ensure there is clear communication warning users that operators need to be trusted.

Invalid state after `swap`

The `DefiEdgeStrategy` method `swap` deletes all data from `ticks`.

However the contract isn't put on hold after that. It means users are not blocked from calling `mint` which would assume the `tick[0]` is valid.

As `swap` can only be called by the *operator*, this isn't exploitable by an unprivileged attacker.

Suggested Remedy: either force `swap` to put the contract on hold, or validate that `ticks.length > 0` in `mint`.

UPDATE: the development team provided a fix for this issue in commit `04865db27600f8b282700c3c4227d0191073c092`. We reviewed the fix and it seems to correctly solve the issue.

Missed Vulnerabilities

For completeness, we'll mention here a vulnerabilities missed in the original audit which was identified by the development team afterwards.

Stealing of funds due to accounting error in minting/burning

The issue was caused by an accounting error in `UniswapPoolActions.sol`, `DefiEdgeStrategy.sol`, and `BaseStrategy.sol` whereas the minting and burning of shares didn't consider the accumulated (and not yet claimed) fees in the calculation of amounts burned and minted. This could lead to an attacker minting and

then burning to obtain tokens in excess of resources used, resulting in theft of funds.

This issue was **fixed** in commit `04865db27600f8b282700c3c4227d0191073c092`. We reviewed the fix, and it seems to correctly solve the problem.

New issues introduced in the updated version

This section will discuss some issues introduced in the new updated version of the code, found either by the authors of this audit, or reported to us by the development team.

Some suggestions for mitigation are offered.

Incorrect Calculation of returned tokens in "Burn"

A fix to accounting issues in `mint` / `burn` introduced new accounting errors, that lead to the amount of tokens obtained from a `burn` being understimated.

1 - When calling "burnLiquidity" the amount of liquidity burned is calculated using the fraction of the shares: https://github.com/unbound-finance/defiedge-core/blob/353bc236e82fdfb672208e551b2465f258922336/contracts/base/UniswapV3LiquidityManager.sol#L99

2 - The code in "burn" adds the amount of tokens burned to the total "collect0" / "collect1". https://github.com/unbound-finance/defiedge-core/blob/353bc236e82fdfb672208e551b2465f258922336/contracts/DefiEdgeStrategy.sol#L191

3 - The total "collect0" / "collect1" is fractioned based on the number of shares: https://github.com/unbound-finance/defiedge-core/blob/353bc236e82fdfb672208e551b2465f258922336/contracts/DefiEdgeStrategy.sol#L205

That means the number of tokens obtained for burning liquidity is fractioned twice: that means it will be proportional to the square of the % of shares being burned.

Consequently, users calling `burn` would lose funds.

Suggestion: the easiest way to calculate the amount of tokens to return would be to calculate the total amount of tokens owned by the contract and the corresponding fraction *after* burning liquidity from the pool and collecting fees.

Manipulation of Liquidity distribution and possible theft of Funds

After a major refactoring, the ability for users to decide in which "tick" to mint liquidity when adding funds to the strategy was added.

By selecting the "tick id" a user calling `mint` can choose to have liquidity corresponding to the funds he added deposited to any of registered ticks in the UniswapV3 Pool, or to have funds sit idly in the strategy contract itself.

However, `burn` is still agnostic to that choice made by the user, and will burn liquidity from *all* ticks in order to withdraw funds to the user.

That means any user can manipulate the distribution of *other user's* funds in the various ticks by calling `mint` and `burn` immediately. See the example

```
Let's suppose there are two valid ticks, "tick0" and "tick1", and that
the strategy has a value of 10 in tick0 and nothing in tick1.

If the attacker now mints a value of 100, choosing tick1, the strategy
will have 10 in tick0 and 100 in tick1, while the attacker has 90% of
shares.

If now the attacker burns all shares, he will get back 100 value,
while the strategy will have 1 in tick0 and 9 in tick1.
```

```
That means the attacker moved 90% of the strategy's tokens from tick0
to tick1, without spending any tokens.
```

The same could be done to transfer it to a balance in the strategy (not in any pool), by choosing tick "-1". see: https://github.com/unbound-finance/defiedge-core/blob/353bc236e82fdfb672208e551b2465f258922336/contracts/DefiEdgeStrategy.sol#L87

This can lead to an attack to steal funds by manipulating the UniswapV3Pool prices, then adding liquidity.

1 - The attacker moves most of the liquidity in the strategy outside of the pool (tick "-1"), using the method above.

2 - Attacker buys token0, moving price up in the pool.

3 - Attacker now forces the strategy to move it's balance into tick0 (price is within tick0).

4 - Attacker sells token1. Now slippage is lower due to added liquidity, so there's a profit.

5 - Repeat until all liquidity in the strategy is depleted.

While the addition of manipulation checks on the UniswapV3 Pool (by comparing deviation from oracle price) reduces how much can be obtained in one iteration of the attack, it can't prevent the attack from happening -- unless the maximum price deviation is so small that it might become impractical for the normal operation of the strategy.

Suggested Mitigation: remove the possibility of users choosing the tick for depositing liquidity. Instead have the tokens be distributed between all ticks according to ratios set by the operator during rebase. If that isn't possible, at very least a delay should be added between `mint` and `burn` so as to prevent the use of flash loans to manipulated the distribution of liquidity between ticks.

Manipulation of UniswapV3Pool by Operator

Even if users in general aren't able to manipulate the distribution of liquidity between ticks, the operator role is still capable of performing the attack above by rebasing the strategy multiple times.

While the operator must be a trusted role (at least to some extent, see above), it might be a good idea to add some further mitigation for this risk.

A member of the development team suggested introducing a rate limit in calls to rebase. It is our opinion that change could considerably mitigate the risk of theft of funds by an operator -- but there are still other ways of manipulating the strategy that might not require multiple calls to `rebase`.

While any effort to limit the ability of a trusted actor to "pull the rug" is welcome, we still recommend clear communication about the trusted role of operators.

Conclusion

This document described the methodology for conducting the audit of the code, the potential attacks and weaknesses considered, as well as limitations of this report.

Issues found were reported and remedies suggested.

The development team resolved the mentioned issues and reported an additional one which was resolved as well.

New issues introduced after a re-structuring of the contracts were presented, and suggestions for mitigation presented.