DefiEdge TWAP

Security Assessment

October 17th, 2022

Prepared By:

Riley Holterhus - Independent Consultant

Table of Contents

Issues

Report Version

Version	Date	Description
1.0	October 5th, 2022	Initial Findings Report
1.1	October 17th, 2022	Final Report

Disclaimer

This report does not indicate the endorsement of any project, individual or entity. This report may not be inclusive of all vulnerabilities within the scope of the review. The author of this report does not take any responsibility for any damage or harm caused by using the information in this document. By using the information in this report, you are doing so at your own risk. This report has been provided on an "as-is" basis and is **NOT A GUARANTEE OR WARRANTY OF ANY FORM.**

Summary

From September 26th, 2022, through October 5th 2022, DefiEdge contracted Riley Holterhus to independently review their TWAP codebase. Over the course of those 1.5 weeks, the relevant code was manually analyzed for various security issues and logic flaws. In total, there were 3 high severity, 3 medium severity, 2 low severity and 6 informational issues brought to the team's attention.

The codebase can be found in DefiEdge's github repository: https://github.com/unbound-finance/defiedge-core/tree/main/contracts/twap (private as of October 5th, 2022). The following files were the scope of the review, all at commit a0eb9b04ab375cda8c26d23df53f235760603d4c:

- base/TwapStrategyBase.sol
- base/TwapStrategyManager.sol
- base/UniswapV3TwapLiquidityManager.sol
- libraries/TwapOracleLibrary.sol
- libraries/TwapShareHelper.sol
- DefiEdgeTwapStrategy.sol
- DefiEdgeTwapStrategyDeployer.sol
- DefiEdgeTwapStrategyFactory.sol

[H-01] Strategy manager can drain entire TVL

Description: Each DefiEdge strategy exposes a `swap` function that can be called by the strategy manager. This function allows the manager to swap amounts of `token0` for `token1` (and vice versa) by specifying some calldata to be sent to the `oneInchRouter`. The OneInchHelper library (out of scope) attempts to vet the calldata that the strategy manager calls `swap` with. However, it is quite easy for the strategy manager to take over control flow in the middle of the call to the `oneInchRouter`. After they take over control flow, the strategy manager can deposit the strategy's funds back into the protocol by calling `mint`. Since the funds are returned to the strategy this way, the slippage checks are tricked into thinking very few tokens have left the protocol. The strategy manager is essentially minting free shares to themselves with this attack, and they can repeat this until they are able to withdraw the entire TVL of the strategy.

```
function swap(bytes calldata data) public onlyOperator {
    LocalVariables_Balances memory balances;

    (IERC20 srcToken, IERC20 dstToken, uint256 amount) = OneInchHelper
        .decodeData(IERC20(token0), IERC20(token1), data);

    require(
        (srcToken == token0 && dstToken == token1) ||
            (srcToken == token1 && dstToken == token0),
        "IA"
    );

    balances.tokenInBalBefore = srcToken.balanceOf(address(this));
    balances.tokenOutBalBefore = dstToken.balanceOf(address(this));

    srcToken.safeIncreaseAllowance(address(oneInchRouter), amount);

    // Interact with 1inch through contract call with data
    (bool success, bytes memory returnData) = address(oneInchRouter).call{
        value  0
    }(data);

    // Verify return status and data
    if (!success) {
        uint256 length = returnData.length;
        if (length < 68) {
            // If the returnData length is less than 68 the transaction failed
            revert("swap");
        } else {
            // Look for revert reason and bubble it up if present
            uint256 t;
            assembly {
                returnData  = add(returnData, 4)
                t  = mload(returnData) // Save the content of the length slot
                mstore(returnData, sub(length, 4)) // Set proper length
            }
            string memory reason = abi.decode(returnData, (string));
            assembly {
                mstore(returnData, t) // Restore the content of the length slot
            }
            revert(reason);
        }
    }

    balances.tokenInBalAfter = srcToken.balanceOf(address(this));
    balances.tokenOutBalAfter = dstToken.balanceOf(address(this));

    uint256 amountIn = balances.tokenInBalBefore.sub(
        balances.tokenInBalAfter
    );
    uint256 amountOut = balances.tokenOutBalAfter.sub(
        balances.tokenOutBalBefore
    );

    // check if swap exceed allowed deviation and revert if maximum swap reached
    if (
        TwapOracleLibrary.isSwapExceedDeviation(
            factory,
            pool,
            chainlinkRegistry,
            amountIn,
            amountOut,
            address(srcToken),
            address(dstToken),
            manager,
            useTwap
        )
    ) {
        manager.increamentSwapCounter();
    }

    require(
        TwapOracleLibrary.allowSwap(
            pool,
            factory,
            amountIn,
            amountOut,
            address(srcToken),
            address(dstToken),
            manager,
            useTwap
        ),
        "S"
    );
}
```

Proof of Concept:

- The TVL of the strategy is 500 USDC and 500 DAI
- Strategy manager makes all TVL liquid by calling `rebalance` with `_burnAll = true`
- Strategy manager calls `swap` w/ `data =”swap(caller=attacker,...)”`
- In the swap, the strategy manager gains control flow and has been transferred all 500 USDC
- They deposit 499 UDSC back into strategy with `mint`, then transfer 1 DAI to the strategy
- The `swap` call regains control flow. The strategy thinks 1 USDC was swapped for 1 DAI
- Strategy manager withdraws the free shares they were minted, and can repeat to drain TVL

Recommendation: Add reentrancy guards to all external/public functions in the strategy. This would prevent the strategy manager from calling `mint` in the middle of a `swap` call. Also, add an extra check in `swap` to ensure the total supply of strategy shares does not change before and after the `oneInchRouter` call.

Status: Fixed in commit 5baad29010418239f2812a116a4cf2ab0c2bcae6.

[H-02] Exploitable reentrancy in burn function

Description: When a user wants to withdraw their tokens from a strategy, they call the `burn` function. This function calculates the number of tokens the user is owed based on the number of shares being burned. Part of this calculation is based on the unused token balance of the protocol (lines 203-204 below). At the end of the function, the tokens are transferred to the user. There are some ERC20 compliant tokens that allow the receiver of a transfer to gain control flow (e.g. ERC777 tokens). If `token0` is one of these tokens, then an attacker could potentially reenter the `burn` function before the `token1` balance decreases. The attacker could leverage this unexpected state to steal funds that do not belong to them.

```
172 function burn(
173     uint256 _shares,
174     uint256 _amount0Min,
175     uint256 _amount1Min
176 ) external returns (uint256 collect0, uint256 collect1) {
177     // check if the user has sufficient shares
178     require(balanceOf(msg.sender) >= _shares && _shares != 0, "INS");
179 
180     uint256 amount0;
181     uint256 amount1;
182 
183     // burn liquidity based on shares from existing ticks
184     for (uint256 i = 0; i < ticks.length; i++) {
185         Tick storage tick = ticks[i];
186 
187         uint256 fee0;
188         uint256 fee1;
189         // burn liquidity and collect fees
190         (amount0, amount1, fee0, fee1) = burnLiquidity(
191             tick.tickLower,
192             tick.tickUpper,
193             _shares,
194             0
195         );
196 
197         // add to total amounts
198         collect0 = collect0.add(amount0);
199         collect1 = collect1.add(amount1);
200     }
201 
202     // give from unused amounts
203     uint256 total0 = IERC20(token0).balanceOf(address(this));
204     uint256 total1 = IERC20(token1).balanceOf(address(this));
205 
206     uint256 _totalSupply = totalSupply();
207 
208     if (total0 > collect0) {
209         collect0 = collect0.add(
210             FullMath.mulDiv(total0 - collect0, _shares, _totalSupply)
211         );
212     }
213 
214     if (total1 > collect1) {
215         collect1 = collect1.add(
216             FullMath.mulDiv(total1 - collect1, _shares, _totalSupply)
```

```
        );
    }

    // check slippage
    require(_amount0Min <= collect0 && _amount1Min <= collect1, "S");

    // burn shares
    _burn(msg.sender, _shares);

    // transfer tokens
    if (collect0 > 0) {
        TransferHelper.safeTransfer(address(token0), msg.sender, collect0);
    }
    if (collect1 > 0) {
        TransferHelper.safeTransfer(address(token1), msg.sender, collect1);
    }

    emit Burn(msg.sender, _shares, collect0, collect1);
}
```

Proof of Concept:

- Strategy has 100 USDC + 100 DAI, all liquid. Attacker has 100 shares (out of 200 totalSupply)
- Attacker calls `burn` with 10 shares and is first transferred 5 USDC. Pretend this has a callback
- Attacker reenters `burn` with their remaining 90 shares (totalSupply is 190 at this point)
- Attacker receives (90/190)*100 = 47 DAI and (90/190)*95 = 45 USDC
- Inner call ends, attacker gets 5 DAI from the first `burn` of 10
- End result: attacker receives 50 USDC + 52 DAI, more than expected. Attack can be repeated

Recommendation: Add reentrancy checks to all external/public functions in the strategy. This also is required to mitigate finding [H-01].

Status: Fixed in commit 5baad29010418239f2812a116a4cf2ab0c2bcae6.

[H-03] Strategy manager slippage abuse

Description: For each pool there exists an `allowedSwapDeviation` value and an `allowedSlippage` value. The strategy manager can never perform a `swap` that breaks the threshold of the `allowedSlippage`, and the strategy manager is limited in the number of swaps each day that breaks the `allowedSwapDeviation` threshold. However, even if these values are quite small, the strategy manager can still perform as many swaps as they want with slippage that is *just under* the amount required to trigger any checks. The strategy manager would stand to gain from this, as they can simply swap tokens with themselves using the `oneInchRouter` and disguise the theft as a small “slippage”. With enough swaps, the strategy manager could drain a large portion of the strategy’s TVL.

```
function swap(bytes calldata data) public onlyOperator {
    LocalVariables_Balances memory balances;

    (IERC20 srcToken, IERC20 dstToken, uint256 amount) = OneInchHelper
        .decodeData(IERC20(token0), IERC20(token1), data);

    require(
        (srcToken == token0 && dstToken == token1) ||
            (srcToken == token1 && dstToken == token0),
        "IA"
    );

    balances.tokenInBalBefore = srcToken.balanceOf(address(this));
    balances.tokenOutBalBefore = dstToken.balanceOf(address(this));

    srcToken.safeIncreaseAllowance(address(oneInchRouter), amount);

    // Interact with 1inch through contract call with data
    (bool success, bytes memory returnData) = address(oneInchRouter).call{
        value  0
    }(data);

    // Verify return status and data
    if (!success) {
        uint256 length = returnData.length;
        if (length < 68) {
            // If the returnData length is less than 68, the transaction failed
            revert("swap");
        } else {
            // Look for revert reason and bubble it up if present
            uint256 t;
            assembly {
                returnData  = add(returnData, 4)
                t  = mload(returnData) // Save the content of the length slot
                mstore(returnData, sub(length, 4)) // Set proper length
            }
            string memory reason = abi.decode(returnData, (string));
            assembly {
                mstore(returnData, t) // Restore the content of the length slot
            }
            revert(reason);
        }
    }

    balances.tokenInBalAfter = srcToken.balanceOf(address(this));
309      balances.tokenOutBalAfter = dstToken.balanceOf(address(this));
310
311      uint256 amountIn = balances.tokenInBalBefore.sub(
312          balances.tokenInBalAfter
313      );
314      uint256 amountOut = balances.tokenOutBalAfter.sub(
315          balances.tokenOutBalBefore
316      );
317
318      // check if swap exceed allowed deviation and revert if maximum swap reached
319      if (
320          TwapOracleLibrary.isSwapExceedDeviation(
321              factory,
322              pool,
323              chainlinkRegistry,
324              amountIn,
325              amountOut,
326              address(srcToken),
327              address(dstToken),
328              manager,
329              useTwap
330          )
331      ) {
332          manager.increamentSwapCounter();
333      }
334
335      require(
336          TwapOracleLibrary.allowSwap(
337              pool,
338              factory,
339              amountIn,
340              amountOut,
341              address(srcToken),
342              address(dstToken),
343              manager,
344              useTwap
345          ),
346          "S"
347      );
348  }
```

Proof of Concept:

- For a given pool, `allowedSwapDeviation` and `allowedSlippage` are both at least 0.1%
- Strategy manager performs 250 “swaps” with themself, each with slippage 1 wei less than 0.1%
- After this is done the protocol has 0.999^250 ≈ 78% of its funds remaining, so loss of 22%
- Strategy manager can steal more with >250 swaps, maybe atomically using a malicious contract

Recommendation: Limit the number of swaps the manager can do in a day regardless of the `swapDeviation`. In other words, delete the `if` statement on lines 319-330, and increment the manager’s `swapCounter` on every swap. Also, make sure that the `slippage` allowance and the number of allowed swaps are sufficiently low. A slippage of 0.5% (the default for uniswap swaps) and a maximum of 10 swaps per day would limit the theft to at most 1-0.995^10 ≈ 5% per day. In the absolute worst case of a quick theft, the strategy manager would be able to steal 5% of the TVL at the very end of one day and then another 5% right after (since it would be a new day). It would be beneficial if DefiEdge did some monitoring for malicious strategy managers that attempt to do this attack, so that users can be informed to withdraw funds. Also, add the `onlyValidStrategy` modifier to the `swap` function, so that this attack can be stopped by the governance once the strategy manager is caught.

Status: Fixed in commit 7fe394557c34ae1776c2b87d891ba22ab1ba6f26 (made increment happen on every swap) and commit 4666313d6aa31a6acefe7b54df65f1c627ac4600 (added `onlyValidStrategy` modifier).

[M-01] Default TWAP oracle period is too short

Description: DefiEdge utilizes the UniswapV3 TWAP oracle from the strategy's associated pool in two different ways – to initially price the token shares, and to calculate slippage when TVL is swapped between the two tokens. For each pool, there is a `twapPricePeriod` that can be configured by the governance. If a pool does not have a specified `twapPricePeriod`, the protocol defaults to a period of 20 seconds. This value is a very small default and would likely leave some strategies susceptible to TWAP oracle attacks. Luckily, the only person that benefits from this attack would be the strategy manager (they are the only ones that can call `swap`). It is still important to prevent the strategy manager from "rug-pulling", so this should be fixed.

```
uint256 public override defaultTwapPricePeriod = 20;
```

Proof of Concept:

- Strategy manager manipulates a pool's spot price at the end of block N (using their own capital)
- Strategy manager calls `swap` at the top of block N+1
- Slippage calculations use manipulated TWAP, strategy manager uses this to steal funds
- Immediately after, strategy manager undoes their original manipulation
- Strategy manager has potentially made a profit now (could depend on liquidity concentration)

Recommendation: Increase the `defaultTwapPricePeriod` to be 600 seconds. This would increase the cost of the above attack by several orders of magnitude.

Status: Fixed in commit 5e2b5cc6d8625a0d608f44465ddba4bdb5a6c006.

[M-02] TWAP observation cardinality may be too small

Description: Uniswap V3 pools initially store TWAP observations for only 1 block. If longer time periods are needed for a pool (which is certainly the case for DefiEdge), then someone needs to call the pool's `increaseObservationCardinalityNext` function. On Polygon/Arbitrum/Optimism there are still many important Uniswap V3 pools that only store the default single observation (for example Polygon Frax-USDC at address 0xbeaf7156ba07c3df8fac42e90188c5a752470db7, which is a pool currently used in an official DefiEdge strategy). On these pools, any attempts to read a TWAP greater than a few seconds in the past will revert, which will break some functionalities of the DefiEdge strategies.

Proof of Concept:

- DefiEdge governance changes the `defaultTwapPeriod` from 600s to 1200s
- A strategy manager tries to call `swap` on their pool, which only has capability for 600s TWAP
- The call will revert until someone calls `increaseObservationCardinalityNext`

Recommendation: Keep in mind that by default new Uniswap V3 pools will be incompatible with DefiEdge. One possible fix is to include calls to `increaseObservationCardinalityNext` in the DefiEdge protocol itself as needed. This would require a lot of new code, so a simpler option is to just keep it in mind and maybe add it to the DefiEdge UI for strategy managers.

Status: Acknowledged – team will manually increment the pool's observation cardinality as needed and UI will only show pools with large enough cardinalities.

[M-03] Rebalance tick indices can be very error prone

Description: The strategy manager uses the `rebalance` function whenever they want to change the allocation of the strategy's capital to different ticks. In this function, notice that the main loop addresses the tick based on `_existingTicks[i].index`. Also, notice in the second code snippet below that `burnLiquiditySingle` rearranges the order of the elements in the storage array `ticks` to delete elements (lines 255-258). This makes `rebalance` very error prone for the strategy manager, as the tick at index `i` at the start of the main loop will not necessarily be the same tick in the middle of the loop. It would be very easy for a strategy manager to not realize this strange behavior and accidentally allocate more/less funds then they wanted to certain ticks.

```
244  function rebalance(
245      bytes calldata _swapData,
246      PartialTick[] calldata _existingTicks,
247      NewTick[] calldata _newTicks,
248      bool _burnAll
249  ) external onlyOperator onlyValidStrategy {
250      if (_burnAll) {
251          require(_existingTicks.length == 0, "IA");
252          onHold = true;
253          burnAllLiquidity();
254          delete ticks;
255          emit Hold();
256      }
257
258      //swap from 1inch if needed
259      if (_swapData.length > 0) {
260          swap(_swapData);
261      }
262
263      // redeploy the partial ticks
264      if (_existingTicks.length > 0) {
265          for (uint256 i = 0; i < _existingTicks.length; i++) {
266              Tick memory _tick = ticks[_existingTicks[i].index];
267
268              Tick storage tick;
269
270              if (_existingTicks[i].burn) {
271                  // burn liquidity from range
272                  burnLiquiditySingle(_existingTicks[i].index);
273              } else {
274                  tick = ticks[_existingTicks[i].index];
275              }
276
277              if (
278                  _existingTicks[i].amount0 > 0 ||
279                  _existingTicks[i].amount1 > 0
280              ) {
281                  // mint liquidity
282                  mintLiquidity(
283                      _tick.tickLower,
284                      _tick.tickUpper,
285                      _existingTicks[i].amount0,
286                      _existingTicks[i].amount1,
287                      address(this)
288                  );
```

```

                if (_existingTicks[i].burn) {
                    // push to ticks array
                    ticks.push(Tick(_tick.tickLower, _tick.tickUpper));
                }
            }
        }

        emit PartialRebalance(_existingTicks);
    }

    // deploy liquidity into new ticks
    if (_newTicks.length > 0) {
        redeploy(_newTicks);
        emit Rebalance(_newTicks);
    }

    require(!isInvalidTicks(ticks), "IT");
    // checks for valid ticks length
    require(ticks.length <= MAX_TICK_LENGTH + 10, "ITL");
}
```

```
function burnLiquiditySingle(uint256 _tickIndex)
    public
    returns (
        uint256 amount0,
        uint256 amount1,
        uint256 fee0,
        uint256 fee1
    )
{
    require(manager.isAllowedToBurn(msg.sender), "N");

    Tick storage tick = ticks[_tickIndex];

    (uint128 currentLiquidity, , , , ) = pool.positions(
        PositionKey.compute(address(this), tick.tickLower, tick.tickUpper)
    );

    if (currentLiquidity > 0) {
        (amount0, amount1, fee0, fee1) = burnLiquidity(
            tick.tickLower,
            tick.tickUpper,
            0,
            currentLiquidity
        );
    }

    // shift the index element at last of array
    ticks[_tickIndex] = ticks[ticks.length - 1];
    // remove last element
    ticks.pop();
}
```

Proof of Concept:

- There are three elements of the `ticks` array
- Strategy manager wants to burn all liquidity from ticks at index 0 and 2
- Strategy manager calls `rebalance` with `_existingTicks` using indices 0, 2 in that order
- Call reverts since after first loop iteration, index 2 doesn’t even exist anymore

- Strategy manager could have instead messed up and sent more/less liquidity to wrong ticks

Recommendation: Refactor the code so that the indices do not change their meaning in the middle of the `rebalance` loop. An implementation that deletes all the appropriate ticks at the end of `rebalance` would be much easier to understand.

Status: Fixed in commits 8c41d830d267844ed0eddb92b21b9c9b53dd8d71 and cdd3dad74acbb2adc5280d890b202eb985bc3b96. The code now requires that `_existingTicks` be provided decreasing in `index`. Since the swap + pop method of deleting elements from the array does not interfere with indices lower than the element you are deleting, this solves the problem.

[L-01] Early user can inflate share prices >$100/share

Description: DefiEdge strategies do some initial share calculations to make the strategy start off with a price of $100 per share of the strategy. If the first user of the strategy initially mints a small amount of shares, they can directly transfer some tokens to the strategy to inflate the share price greatly. This could look strange on the UI, and perhaps could be abused by strategy managers into giving the illusion that their strategy is very valuable when it really isn't.

Proof of Concept:

- First user decides to provide $1 of liquidity to the strategy
- Next, the first user directly transfers an additional $1 of tokens directly to the strategy
- The share price is now $200 per share, and when users see this on the UI they might be mislead

Recommendation: Be aware that the initial users have some control over the share price of a DefiEdge strategy. Also, consider putting some sort of warning on the UI for strategies that have low TVL but high price per share.

Status: Acknowledged.

[L-02] Strategy factory variables not always validated

Description: The DefiEdge strategy factory has some variables that are validated in the constructor to be less than some threshold. For example, `defaultAllowedSlippage` and `defaultAllowedSwapDeviation` are initially validated to be at most 10%. However, it is possible for the governance to update these values later on (e.g. by using the `changeDefaultValues` function), and in these cases there are no such validation checks.

Proof of Concept:

- Strategy factory initially has `defaultAllowedSlippage` as 0.5%, less than the 10% threshold
- Governance calls `changeDefaultValues` to change `defaultAllowedSlippage` to 100%
- This does not revert, but 100% would have reverted as an initial value in the constructor

Recommendation: Consider adding validation checks to all setter functions in the strategy factory.

Status: Acknowledged. The initial constructor validations were removed to make it more consistent.

Informational Findings

1. **Finding:** In `TwapOracleLibrary.sol` the function `getUniswapPrice` is described in the comments to get the "latest Uniswap price in the pool, price of token1 represented in token0". This comment could be misleading as Uniswap pools are priced in terms of token1. Consider changing the comment to "price of token0 in terms of token1".
   Status: Fixed in commit bc3edb58640ee13c2de7c45498903bfa595a0cdc.

2. **Finding:** In various places in a strategy, the length two boolean array `useTwap` is used to determine whether to use the TWAP oracle for pricing tokens. However, the first value of the array is the only value that is ever used, and the second value is supposed to be opposite boolean as the first value. Consider changing the array to be a single boolean variable instead.
   Status: Acknowledged.

3. **Finding:** If someone calls `getAUMWithFees`, then all the fees are collected before a single call to `_transferPerformanceFees` is made. On the other hand, in the `burn` function, `_transferPerformanceFees` is called multiple times (because `burnLiquidity` is called in a loop). Calling this helper function multiple times in a transaction will lead to much higher gas costs, since there will be many more external calls. Consider aggregating the fees and sending once at the end, like how `getAUMWithFees` does it.
   Status: Fixed in commits d5f1960ec080584a92a6b29a161b39cd37a95e75 and a63bc97e1eabcf58d9a9a76e048779115f72f1af.

4. **Finding:** In the strategy's `rebalance` function, there is some logic related to a storage variable `tick`. However, this variable is never used, so consider removing all this logic.
   Status: Fixed in commit 4a1c6926256bfc73e5bae3500cde477b255fb68e.

5. **Finding:** The strategy manager has the `increamentSwapCounter` function that is called from the strategy. This is a small typo and is probably meant to be `incrementSwapCounter`.
   Status: Fixed in commit a97a32f9af5a55301c5aac88fc1c7986f94970e3.

6. **Finding:** The strategy implements the `uniswapV3MintCallback` to provide tokens to the pool for minting. The implementation of this function allows for the `payer` to be someone other than the strategy itself, but this functionality is never used elsewhere in the protocol (every path that results in `uniswapV3MintCallback` being called will have the `payer` be `address(this)`). Consider removing this extra functionality of getting other users to pay.
   Status: Fixed in commit 82d9c2359ab0c4721dbe221f28eef87ab89006cb.