HALBORN

MonoX

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **May 3rd-15th, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	05/13/2021	Piotr Cielas
0.5	Document Edits	05/13/2021	Piotr Cielas
0.6	Document Edits	05/14/2021	Gabi Urrutia
1.0	Final Version	05/17/2021	Piotr Cielas
1.1	Remediation Plan	07/15/2021	Piotr Cielas
1.1	Final Review	07/27/2021	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

MonoX is a new DeFi protocol using a single token design for liquidity pools (instead of using pool pairs). This is made possible by grouping deposited tokens into a virtual pair with the vUSD stablecoin.

MonoX engaged Halborn to conduct a security assessment on their smart contracts beginning on May 3rd, 2021 and ending May 15th, 2021. The security assessment was scoped to smart contracts implementing the core protocol and the staking mechanism, and an audit of the security risk and implications regarding the changes introduced by the development team at MonoX prior to its production release shortly following the assessments deadline.

Though this security audit's outcome is satisfactory, only the most essential aspects were tested and verified to achieve objectives and deliverables set in the scope due to time and resource constraints. It is essential to note the use of the best practices for secure smart-contract development.

1.2 AUDIT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned one full time security engineer to audit the security of the assets in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit to achieve the following:

- Ensure that smart contract functions are intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified few security risks, and recommends performing further testing to validate extended safety and correctness in context to the whole set of contracts. External threats, such as economic attacks, oracle attacks, and inter-contract functions and calls should be validated for expected logic and state.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process,and implementation; automated testing techniques help enhance coverage of the smart contract code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough
- Graphing out functionality and contract logic/connectivity/functions(solgraph)
- Manual Assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes (Hardhat and manual deployments on Ganache)
- Manual testing with custom Javascript.
- Static Analysis of security for scoped contract, and imported functions.(Slither)
- Scanning of solidity files for vulnerabilities, security hotspots or bugs. (MythX)
- Testnet deployment (Remix IDE)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident, and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. It's quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that was used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

IN-SCOPE:

The security assessment was scoped to the smart contracts:

Monoswap Core:

- Monoswap.sol
- MonoXPool.sol
- VUSD.sol

commit #c1e16f0b588aeb129d8e13abbc9d39ab3a3392c3

Monoswap Staking:

- MonoswapStaking.sol
- MonoToken.sol

commit #89115cd39237c496b60e8a71b07f46968bd854f2

OUT-OF-SCOPE:

Dependencies and external libraries.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	3	1	5	2

LIKELIHOOD

IMPACT

(HAL-08)			(HAL-01) (HAL-02) (HAL-03)	
	(HAL-05) (HAL-06)	(HAL-04)		
	(HAL-07) (HAL-09)			
(HAL-10) (HAL-11)				

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
UNRESTRICTED POOL TOKEN MINTING	High	SOLVED 5/24/2021
POOL BLOCKING	High	SOLVED 5/24/2021
ROLE-BASED ACCESS CONTROL MISSING	High	SOLVED 5/24/2021
INTEGER OVERFLOW	Medium	SOLVED 6/2/2021
EXTERNAL FUNCTION CALLS WITHIN LOOP	Low	SOLVED 6/1/2021
DIVIDE BEFORE MULTIPLY	Low	SOLVED 7/15/2021
ADDRESS VALIDATION MISSING	Low	PARTIALLY SOLVED 5/24/2021
USE OF BLOCK.TIMESTAMP	Low	SOLVED 6/2/2021
TAUTOLOGY EXPRESSIONS	Low	SOLVED 5/24/2021
POSSIBLE MISUSE OF PUBLIC FUNCTIONS	Informational	PARTIALLY SOLVED 5/24/2021
IMPRECISION OF A CONSTANT	Informational	ACKNOWLEDGED 7/15/2021

FINDINGS & TECH DETAILS

3.1 (HAL-01) UNRESTRICTED POOL TOKEN MINTING - HIGH

Description:

One of MonoX's main objectives is to allow users for listing ERC20 tokens without the need for providing liquidity. In order to keep track of users' shares in pools, a corresponding amount of liquidity pool tokens is minted to providers. The exact amount to be minted depends on e.g. the declared amount of ERC20 tokens added to the pool and the token price, intially set by the provider.

In the addLiquidityPair function MonoX use OpenZeppelin's safeTransferFrom to handle the token transfer. This function calls transferFrom in the token contract to actually execute the transfer. However, since the actual amount transferred ie. the delta of previous (before transfer) and current (after transfer) balance is not verified, a malicious user may list a custom ERC20 token with the transferFrom function modified in such a way that it e.g. does not transfer any tokens at all and the attacker is still going to have their liquidity pool tokens minted anyway.

Code Location:

Attacker-controlled example ERC20 token contract

Listing 1: EvilERC20.sol (Lines 10)

```
function transferFrom(
    address from,
    address to,
    uint256 value
  )
    public
    override
    returns (bool)
  {
    value = 1;
```

```
    require(value <= _balances[from]);
    require(value <= _allowed[from][msg.sender]);
    require(to != address(0));

    _balances[from] = _balances[from].sub(value);
    _balances[to] = _balances[to].add(value);
    _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(
        value);
    emit Transfer(from, to, value);
    return true;
  }
```

MonoX

Listing 2: Monoswap.sol (Lines 235,236)

```
231 function listNewToken (address _token, uint112 _price,
232     uint256 vusdAmount,
233     uint256 tokenAmount,
234     address to) public returns(uint _pid, uint256 liquidity) {
235     _pid = _createPool(_token, _price, PoolStatus.LISTED);
236     liquidity = addLiquidityPair(_token, vusdAmount, tokenAmount,
        to);
237 }
```

Listing 3: Monoswap.sol (Lines 253,258,261,164,267)

```
251     _mintFee(pool.pid, pool.lastPoolValue, poolValue);
252     uint256 _totalSupply = monoXPool.totalSupplyOf(pool.pid);
253     IERC20(_token).safeTransferFrom(msg.sender, address(monoXPool)
          , tokenAmount);
254     if(vusdAmount>0){
255       vUSD.safeTransferFrom(msg.sender, address(monoXPool),
          vusdAmount);
256     }
257
258     uint256 liquidityVusdValue = vusdAmount.add(tokenAmount.mul(
        pool.price)/1e18);
259
260     if(_totalSupply==0){
261       liquidity = liquidityVusdValue.sub(MINIMUM_LIQUIDITY);
262       mint(owner(), pool.pid, MINIMUM_LIQUIDITY); // sorry, oz
```

```
            doesn't allow minting to address(0)
    }else{
      liquidity = _totalSupply.mul(liquidityVusdValue).div(
            poolValue);
    }

    mint(to, pool.pid, liquidity);
}
```

OpenZeppelin

Listing 4: SafeERC20.sol (Lines 24,25)

```
17 library SafeERC20 {
18     using Address for address;
19
20     function safeTransfer(IERC20 token, address to, uint256 value)
           internal {
21         _callOptionalReturn(token, abi.encodeWithSelector(token.
               transfer.selector, to, value));
22     }
23
24     function safeTransferFrom(IERC20 token, address from, address
           to, uint256 value) internal {
25         _callOptionalReturn(token, abi.encodeWithSelector(token.
               transferFrom.selector, from, to, value));
26     }
```

Recommendation:

Whenever tokens are transferred, the delta of the previous (before transfer) and current (after transfer) token balance should be verified to match the user-declared token amount.

Remediation Plan:

SOLVED: Fixed in commit #635a4cee2f2e50d854e06cac47c48aa0fafde2b0. The amount to be minted is calculated now based on the delta of account balance before and after transfer.

3.2 (HAL-02) POOL BLOCKING - HIGH

Description:

One of MonoX's main objectives is to allow users for listing ERC20 tokens without the need for providing liquidity. Users can set arbitrary prices for tokens they list because the Monoswap.sol contract does not verify them against third-party data sources. The price of a given token can be updated only if it has not been swapped for at least 6000 blocks since the last exchange. In consequence, since the contract does not enforce minimum or maximum transaction amount, a malicious user can list tokens, price them way above market rate and keep the price on that level by doing microexchanges once every 6000 blocks thus effectively DoSing the pool.

Code Location:

Listing 5: Monoswap.sol (Lines 235)

```
231 function listNewToken (address _token, uint112 _price,
232     uint256 vusdAmount,
233     uint256 tokenAmount,
234     address to) public returns(uint _pid, uint256 liquidity) {
235     _pid = _createPool(_token, _price, PoolStatus.LISTED);
236     liquidity = addLiquidityPair(_token, vusdAmount, tokenAmount,
            to);
237 }
```

Listing 6: Monoswap.sol (Lines 189,196)

```
177 function _createPool (address _token, uint112 _price, PoolStatus
        _status) lock internal returns(uint256 _pid)  {
178     require(tokenPoolStatus[_token]==0, "Monoswap: Token Exists");
179     require (_token != address(vUSD), "Monoswap: vUSD pool not
            allowed");
180     _pid = poolSize;
181     pools[_token] = PoolInfo({
182       token: _token,
183       pid: _pid,
```

```
184         vusdCredit: 0,
185         vusdDebt: 0,
186         tokenBalance: 0,
187         lastPoolValue: 0,
188         status: _status,
189         price: _price
190       });
191 
192       poolSize = _pid.add(1);
193       tokenPoolStatus[_token]=1;
194 
195       // initialze pool's lasttradingblocknumber as the block number on which the pool is created
196       lastTradedBlock[_token] = block.number;
197 }
```

Listing 7: Monoswap.sol (Lines 463)

```
454 function swapExactTokenForToken(
455     address tokenIn,
456     address tokenOut,
457     uint amountIn,
458     uint amountOutMin,
459     address to,
460     uint deadline
461   ) external virtual ensure(deadline) returns (uint amountOut) {
462     amountOut = swapIn(tokenIn, tokenOut, msg.sender, to, amountIn);
463     require(amountOut >= amountOutMin, 'Monoswap: INSUFFICIENT_OUTPUT_AMOUNT');
464 }
```

Listing 8: Monoswap.sol (Lines 561)

```
560     // record last trade's block number in mapping: lastTradedBlock
561     lastTradedBlock[_token] = block.number;
```

Listing 9: Monoswap.sol (Lines 163)

```
156 function updatePoolPrice(address _token, uint112 _newPrice) public onlyOwner {
```

```
    require(_newPrice > 0, 'Monoswap: zeroPriceNotAccept');
    require(tokenPoolStatus[_token] != 0, "Monoswap: PoolNotExist"
        );

    PoolInfo storage pool = pools[_token];
    require(pool.price != _newPrice, "Monoswap: SamePriceNotAccept
        ");

    require(block.number > lastTradedBlock[_token].add(6000), "
        Monoswap: PoolPriceUpdateLocked");
    pool.price = _newPrice;
    lastTradedBlock[_token] = block.number;
}
```

Recommendation:

If possible, it's recommended to validate tokens' prices (by the use of oracles) on initial listing and on every subsequent price change in order not to allow for manipulating the exchange by malicious users. Additionally, a minimum/maximum input/output amount of tokens could be enforced.

Reference:

Chainlink Price Oracle

Remediation Plan:

SOLVED: Fixed in commit #635a4cee2f2e50d854e06cac47c48aa0fafde2b0. Contract owner can now pause pools and temporarily disable swapping so that users with the PriceAdjuster role (assigned by the contract owner) can update prices.

3.3 (HAL-03) ROLE-BASED ACCESS CONTROL MISSING - HIGH

Description:

In smart contracts, implementing a correct Access Control policy is an essential step to maintain security and decentralization for permissions on a token. All the features of the smart contract , such as mint/burn tokens and pause contracts are given by Access Control. For instance, Ownership is the most common form of Access Control. In other words, the owner of a contract (the account that deployed it by default) can do some administrative tasks on it. Nevertheless, other authorization levels are required to follow the principle of least privilege, also known as least authority. Briefly, any process, user or program only can access to the necessary resources or information. Otherwise, the ownership role is useful in a simple system, but more complex projects require the use of more roles by using Role-based access control.

Code Location:

Listing 10: Monoswap.sol (Lines 130,134,139,145,156)

```
130 function setFeeTo (address _feeTo) onlyOwner external {
131     feeTo = _feeTo;
132 }
133
134 function setFees (uint16 _fees) onlyOwner external {
135     require(_fees<1e3, "fees too large");
136     fees = _fees;
137 }
138
139 function setDevFee (uint16 _devFee) onlyOwner external {
140     require(_devFee<1e3, "devFee too large");
141     devFee = _devFee;
142 }
143
144 // update status of a pool. onlyOwner.
145 function updatePoolStatus(address _token, PoolStatus _status)
        public onlyOwner {
```

```
    PoolInfo storage pool = pools[_token];
    pool.status = _status;
}

/**
 @dev update pools price if there were no active trading for the last 6000 blocks
 @notice Only owner callable, new price can neither be 0 nor be equal to old one
 @param _token pool identifider (token address)
 @param _newPrice new price in wei (uint112)
*/
function updatePoolPrice(address _token, uint112 _newPrice) public onlyOwner {
    require(_newPrice > 0, 'Monoswap: zeroPriceNotAccept');
```

Recommendation:

It's recommended to use role-based access control based on the principle of least privilege to lock permissioned functions using different roles.

Reference:

Least Privilege Principle

Remediation Plan:

SOLVED: Fixed in commit #635a4cee2f2e50d854e06cac47c48aa0fafde2b0. Several new roles were introduced.

3.4 (HAL-04) INTEGER OVERFLOW - MEDIUM

Description:

An overflow happens when an arithmetic operation reaches the maximum size of a type. For instance, in Monoswap.sol, the getAmountOut method is subtracting fees from a fixed number and may end up overflowing the integer since the resulting value is not checked to be greater or equal 0. In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits -- either larger than the maximum or lower than the minimum representable value.

Code Location:

Listing 11: Monoswap.sol (Lines 638)

```
633 function getAmountOut(address tokenIn, address tokenOut,
634     uint256 amountIn) public view returns (uint256 tokenInPrice,
         uint256 tokenOutPrice,
635     uint256 amountOut, uint256 tradeVusdValue) {
636     require(amountIn > 0, 'Monoswap: INSUFFICIENT_INPUT_AMOUNT');
637 
638     uint256 amountInWithFee = amountIn.mul(1e5-fees)/1e5;
639     address vusdAddress = address(vUSD);
```

Listing 12: Monoswap.sol (Lines 584)

```
579 function getAmountIn(address tokenIn, address tokenOut,
580     uint256 amountOut) public view returns (uint256 tokenInPrice,
         uint256 tokenOutPrice,
581     uint256 amountIn, uint256 tradeVusdValue) {
582     require(amountOut > 0, 'Monoswap: INSUFFICIENT_INPUT_AMOUNT');
583 
584     uint256 amountOutWithFee = amountOut.mul(1e5+fees)/1e5;
585     address vusdAddress = address(vUSD);
```

Recommendation:

It is recommended to use vetted safe math libraries for arithmetic operations consistently throughout the smart contract system

Reference:

Ethereum Smart Contract Best Practices - Integer Overflow and Underflow

Remediation Plan:

SOLVED: MonoX is certain the integers reported will not overflow since the fees variable cannot be assigned value greater than 1e3.

3.5 (HAL-05) EXTERNAL FUNCTION CALLS WITHIN LOOP - LOW

Description:

Calls inside a loop might lead to a denial-of-service attack. In on of the functions discovered there is a for loop on variable pid that iterates up to the poolInfo array length. If this integer is evaluated at extremely large numbers this can cause a DoS.

Code Location:

Listing 13: MonoswapStaking.sol (Lines 241)

```
236 function massUpdatePools() public {
237     uint256 length = poolInfo.length;
238     for (uint256 pid = 0; pid < length; ++pid) {
239         PoolInfo storage pool = poolInfo[pid];
240         if (pool.bActive)
241             updatePool(pid);
242     }
243 }
```

Recommendation:

If possible, use pull over push strategy for external calls.

Remediation Plan:

SOLVED: MonoX is certain the DoS scenario is highly unlikely here since all external calls in this loop are made to MonoX-controlled contracts.

3.6 (HAL-06) DIVIDE BEFORE MULTIPLY - LOW

Description:

Solidity integer division might truncate. As a result, performing multiplication before division can sometimes avoid loss of precision. In this audit, there are multiple instances found where division is being performed before multiplication operation in contract file.

Code Location:

Listing 14: MonoswapStaking.sol (Lines 221)

```
if (user.oldReward > 0) {
    monoReward = monoReward.add(user.oldReward.mul(stakedAmount).
        div(user.amount).mul(1e12));
}
```

Recommendation:

Consider doing multiplication operation before division to prevail precision in the values in non floating data type.

Remediation Plan:

SOLVED: fixed in commit #ac21bee3f7f1d7df3529907b0afb0470b0236d07

3.7 (HAL-07) ADDRESS VALIDATION MISSING - LOW

Description:

Address validation is missing in multiple functions in contracts Monoswap.sol and MonoXPool.sol. This may result with users irreversibly locking their tokens when incorrect address is provided.

Code Location:

Listing 15: Monoswap.sol (Lines 169,173)

```
168 function mint (address account, uint256 id, uint256 amount)
        internal {
169     monoXPool.mint(account, id, amount);
170 }
171
172 function burn (address account, uint256 id, uint256 amount)
        internal {
173     monoXPool.burn(account, id, amount);
174 }
```

Listing 16: MonoXPool.sol (Lines 20)

```
19 constructor (address _WETH) {
20       WETH = _WETH;
21 }
```

Listing 17: MonoXPool.sol (Lines 28,33)

```
26 function mint (address account, uint256 id, uint256 amount) public
        onlyOwner {
27     totalSupply[id]=totalSupply[id].add(amount);
28     _mint(account, id, amount, "");
29 }
30
31 function burn (address account, uint256 id, uint256 amount) public
        onlyOwner {
```

```
    totalSupply[id]=totalSupply[id].sub(amount);
    _burn(account, id, amount);
}
```

Recommendation:

Add proper address validation when assigning a value to a variable from user-supplied data. Better yet, address white-listing/black-listing should be implemented in relevant functions if possible.

Remediation Plan:

PARTIALLY SOLVED: Vulnerable function calls in Monoswap.sol have been removed but address validation is missing in MonoXPool.sol.

3.8 (HAL-08) USE OF BLOCK.TIMESTAMP - LOW

Description:

block.timestamp can be influenced by miners to a certain degree, so the testers should be warned that this may have some risk if miners collude on time manipulation to influence the price oracles.

Code Location:

Listing 18: Monoswap.sol (Lines 86)

```
85 modifier ensure(uint deadline) {
86     require(deadline >= block.timestamp, 'Monoswap: EXPIRED');
87     _;
88 }
```

Recommendation:

Use block.number instead of block.timestamp to reduce the risk of MEV attacks. Check if the timescale of the project occurs across years, days and months rather than seconds. If possible, it is recommended to use Oracles.

Remediation Plan:

SOLVED: MonoX does not require timestamps to be extremely precise here (timescales are greater than 900 seconds)

3.9 (HAL-09) TAUTOLOGY EXPRESSIONS - LOW

Description:

In contract Monoswap.sol, tautology expressions have been detected. Such expressions are of no use since they always evaluate true/false regardless of the context they are used in.

Code Location:

Listing 19: Monoswap.sol (Lines 546)

```
544 if(_poolStatus == PoolStatus.LISTED){
545
546         require (_vusdCredit>=0 && _vusdDebt==0, "Monoswap:
            unofficial pool cannot bear debt");
547 }
```

Recommendation:

Correct the expressions. Since _vusdCredit variable is declared as type uint256, it is always greater or equal to 0.

Remediation Plan:

SOLVED: Tautology Expression was removed in commit #635a4cee2f2e50d854e06cac47c48aa0fa

3.10 (HAL-10) POSSIBLE MISUSE OF PUBLIC FUNCTIONS - INFORMATIONAL

Description:

In public functions, array arguments are immediately copied to memory, while external functions can read directly from calldata. Reading calldata is cheaper than memory allocation. Public functions need to write the arguments to memory because public functions may be called internally. Internal calls are passed internally by pointers to memory. Thus, the function expects its arguments being located in memory when the compiler generates the code for an internal function.
Also, methods do not necessarily have to be public if they are only called within the contract-in such case they should be marked internal.

Code Location:

Listing 20: Monoswap.sol (Lines 145)

```
145 function updatePoolStatus(address _token, PoolStatus _status)
        public onlyOwner {
146     PoolInfo storage pool = pools[_token];
147     pool.status = _status;
148 }
```

Listing 21: Monoswap.sol (Lines 156)

```
156 function updatePoolPrice(address _token, uint112 _newPrice) public
        onlyOwner {
157     require(_newPrice > 0, 'Monoswap: zeroPriceNotAccept');
158     require(tokenPoolStatus[_token] != 0, "Monoswap: PoolNotExist"
            );
159
160     PoolInfo storage pool = pools[_token];
```

Listing 22: Monoswap.sol (Lines 234)

```
231 function listNewToken (address _token, uint112 _price,
232     uint256 vusdAmount,
233     uint256 tokenAmount,
234     address to) public returns(uint _pid, uint256 liquidity) {
235     _pid = _createPool(_token, _price, PoolStatus.LISTED);
236     liquidity = addLiquidityPair(_token, vusdAmount, tokenAmount,
        to);
237   }
238 }
```

Listing 23: MonoswapStaking.sol (Lines 147)

```
143 function set(
144     uint256 _pid,
145     uint256 _allocPoint,
146     bool _withUpdate
147 ) public onlyOwner {
148     if (_withUpdate) {
149         massUpdatePools();
150     }
```

Listing 24: MonoswapStaking.sol (Lines 276)

```
276 function stopPool(uint256 _pid) public onlyOwner {
277     updatePool(_pid);
```

Listing 25: MonoswapStaking.sol (Lines 284)

```
284 function migratePool(uint256 _oldPid, uint256 _newPid) public {
285     PoolInfo storage oldPool = poolInfo[_oldPid];
286     PoolInfo storage newPool = poolInfo[_newPid];
```

Listing 26: MonoswapStaking.sol (Lines 326)

```
326 function deposit(uint256 _pid, uint256 _amount) public {
327     PoolInfo storage pool = poolInfo[_pid];
328     UserInfo storage user = userInfo[_pid][msg.sender];
```

Listing 27: MonoswapStaking.sol (Lines 365)

```
function withdraw(uint256 _pid, uint256 _amount) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
```

Listing 28: MonoswapStaking.sol (Lines 413)

```
function emergencyWithdraw(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
```

Recommendation:

Consider as much as possible declaring external variables instead of public variables. As for best practice, you should use external if you expect that the function will only be called externally and use public if you need to call the function internally. To sum up, all can access to public functions, external functions only can be accessed externally and internal functions can only be called within the contract.

3.11 Remediation Plan

PARTIALLY SOLVED: several functions in MonoswapStaking.sol are still public.

3.12 (HAL-11) IMPRECISION OF A CONSTANT - INFORMATIONAL

Description:

During the audit, it has been observed that integers with scientific notations are directly compared with function arguments.

Code Location:

Listing 29: Monoswap.sol (Lines 135,140)

```
134 function setFees (uint16 _fees) onlyOwner external {
135     require(_fees<1e3, "fees too large");
136     fees = _fees;
137 }
138
139 function setDevFee (uint16 _devFee) onlyOwner external {
140     require(_devFee<1e3, "devFee too large");
141     devFee = _devFee;
142 }
```

Listing 30: Monoswap.sol (Lines 213)

```
205 function _mintFee (uint256 pid, uint256 lastPoolValue, uint256
        newPoolValue) internal {
206
207     uint256 _totalSupply = monoXPool.totalSupplyOf(pid);
208     if(newPoolValue>lastPoolValue && lastPoolValue>0) {
209       // safe ops, since newPoolValue>lastPoolValue
210       uint256 deltaPoolValue = newPoolValue - lastPoolValue;
211
212       // safe ops, since newPoolValue = deltaPoolValue +
            lastPoolValue > deltaPoolValue
213       uint256 devLiquidity = _totalSupply.mul(deltaPoolValue).mul(
            devFee).div(newPoolValue-deltaPoolValue)/1e5;
214       monoXPool.mint(feeTo, pid, devLiquidity);
215     }
216
217 }
```

Also lines #584 and #638 in Monoswap.sol.

Listing 31: Monoswap.sol (Lines 225)

```
220 function getPool (address _token) view public returns (uint256 poolValue,
221     uint256 tokenBalanceVusdValue, uint256 vusdCredit, uint256 vusdDebt) {
222     PoolInfo memory pool = pools[_token];
223     vusdCredit = pool.vusdCredit;
224     vusdDebt = pool.vusdDebt;
225     tokenBalanceVusdValue = uint(pool.price).mul(pool.tokenBalance)/1e18;
226
227     poolValue = tokenBalanceVusdValue.add(vusdCredit).sub(vusdDebt);
228 }
```

Also lines #258, #297, #322, #326, #569, #570, #590, #600, #605, #614, #628, #645, #656, #661, #671, #684, #715 and #767 in Monoswap.sol.

Listing 32: MonoswapStaking.sol (Lines 100)

```
86  function initialize(
87          MonoToken _mono,
88          uint256 _monoPerPeriod,
89          uint256 _blockPerPeriod,
90          uint256 _decay
91      ) public initializer {
92          OwnableUpgradeable.__Ownable_init();
93          __ERC1155Holder_init();
94          mono = _mono;
95          monoPerPeriod = _monoPerPeriod;
96          blockPerPeriod = _blockPerPeriod;
97          decay = _decay;
98          startBlock = block.number;
99          currentPeriod = 0;
100         ratios[currentPeriod] = 1e12;
101         totalAllocPoint = 0;
102 }
```

Also lines #176, #185, #211, #221, #232, #265, #297, #299, #316, #317,

#336, #337, #359, #377, #378 and #389 in MonoswapStaking.sol.

Recommendation:

It is recommended to define precision values as constants at the beginning of contract.

Listing 33: Example.sol

```
uint constant PRECISION3 = 1e3;
uint constant PRECISION5 = 1e5;
uint constant PRECISION18 = 1e18;
```

Listing 34: Example.sol

```
uint constant PRECISION12 = 1e12;
```

Remediation Plan:

ACKNOWLEDGED: MonoX refrain from introducing extra variables as it increases the contract size quite a bit and increase gas usage as well. Therefore they are trying not to have a variable unless it's necessary.

3.13 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Results:

MonoX Core

```
'npx hardhat compile' running
Nothing to compile

        -tradeVusdValue = _getAvgPrice(tokenOutPoolPrice,tokenOutPrice).mul(amountOutWithFee) / 1e18 (contracts/
Monoswap.sol#600)
        -amountIn = tradeVusdValue.mul(tokenInPoolTokenBalance).div(amountIn) (contracts/Monoswap.sol#615)
Monoswap.getAmountIn(address,address,uint256) (contracts/Monoswap.sol#579-630) performs a multiplication on the
result of a division:
        -tradeVusdValue = _getAvgPrice(tokenOutPoolPrice,tokenOutPrice).mul(amountOutWithFee) / 1e18 (contracts/
Monoswap.sol#600)
        -amountIn = tradeVusdValue.mul(1e18).div(_getAvgPrice(tokenInPoolPrice,tokenInPrice)) (contracts/Monoswa
p.sol#628)
Monoswap.getAmountOut(address,address,uint256) (contracts/Monoswap.sol#633-686) performs a multiplication on the
 result of a division:
        -amountInWithFee = amountIn.mul(1e5 - fees) / 1e5 (contracts/Monoswap.sol#638)
        -tradeVusdValue = _getAvgPrice(tokenInPoolPrice,tokenInPrice).mul(amountInWithFee) / 1e18 (contracts/Mon
oswap.sol#656)
Monoswap.getAmountOut(address,address,uint256) (contracts/Monoswap.sol#633-686) performs a multiplication on the
 result of a division:
        -tradeVusdValue = _getAvgPrice(tokenInPoolPrice,tokenInPrice).mul(amountInWithFee) / 1e18 (contracts/Mon
oswap.sol#656)
        -amountOut = tradeVusdValue.mul(tokenOutPoolTokenBalance).div(amountOut) (contracts/Monoswap.sol#672)
Monoswap.getAmountOut(address,address,uint256) (contracts/Monoswap.sol#633-686) performs a multiplication on the
 result of a division:
        -tradeVusdValue = _getAvgPrice(tokenInPoolPrice,tokenInPrice).mul(amountInWithFee) / 1e18 (contracts/Mon
oswap.sol#656)
        -amountOut = tradeVusdValue.mul(1e18).div(_getAvgPrice(tokenOutPoolPrice,tokenOutPrice)) (contracts/Mono
swap.sol#684)
Monoswap.swapIn(address,address,address,address,uint256) (contracts/Monoswap.sol#690-741) performs a multiplicat
ion on the result of a division:
        -oneSideFeesInVusd = tokenInPrice.mul(amountIn.mul(fees) / 2e5) / 1e18 (contracts/Monoswap.sol#715)
Monoswap.swapIn(address,address,address,address,uint256) (contracts/Monoswap.sol#690-741) performs a multiplicat
ion on the result of a division:
        -oneSideFeesInVusd = tokenInPrice.mul(amountIn.mul(fees) / 2e5) / 1e18 (contracts/Monoswap.sol#715)
        -oneSideFeesInVusd = oneSideFeesInVusd.mul(2) (contracts/Monoswap.sol#721)
Monoswap.swapOut(address,address,address,address,uint256) (contracts/Monoswap.sol#745-795) performs a multiplica
tion on the result of a division:
        -oneSideFeesInVusd = tokenInPrice.mul(amountIn.mul(fees) / 2e5) / 1e18 (contracts/Monoswap.sol#767)
Monoswap.swapOut(address,address,address,address,uint256) (contracts/Monoswap.sol#745-795) performs a multiplica
tion on the result of a division:
        -oneSideFeesInVusd = tokenInPrice.mul(amountIn.mul(fees) / 2e5) / 1e18 (contracts/Monoswap.sol#767)
        -oneSideFeesInVusd = oneSideFeesInVusd.mul(2) (contracts/Monoswap.sol#773)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
```

All multiplication on the result of division issues reported by the tool here are false positives.

```
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Reentrancy in Monoswap.addLiquidityPair(address,uint256,uint256,address) (contracts/Monoswap.sol#240-275):
        External calls:
        - _mintFee(pool.pid,pool.lastPoolValue,poolValue) (contracts/Monoswap.sol#251)
                - monoXPool.mint(feeTo,pid,devLiquidity) (contracts/Monoswap.sol#214)
        - IERC20(_token).safeTransferFrom(msg.sender,address(monoXPool),tokenAmount) (contracts/Monoswap.sol#253)
        - vUSD.safeTransferFrom(msg.sender,address(monoXPool),vusdAmount) (contracts/Monoswap.sol#255)
        - mint(owner(),pool.pid,MINIMUM_LIQUIDITY) (contracts/Monoswap.sol#262)
                - monoXPool.mint(account,id,amount) (contracts/Monoswap.sol#169)
        - mint(to,pool.pid,liquidity) (contracts/Monoswap.sol#267)
                - monoXPool.mint(account,id,amount) (contracts/Monoswap.sol#169)
        State variables written after the call(s):
        - _syncPoolInfo(_token,vusdAmount,0) (contracts/Monoswap.sol#268)
                - pools[_token].tokenBalance = uint112(tokenReserve) (contracts/Monoswap.sol#300)
                - pools[_token].lastPoolValue = poolValue (contracts/Monoswap.sol#302)
                - pools[_token].vusdCredit = uint112(_vusdCredit) (contracts/Monoswap.sol#532)
                - pools[_token].vusdDebt = uint112(_vusdDebt) (contracts/Monoswap.sol#533)
                - pools[_token].vusdCredit = uint112(_vusdCredit) (contracts/Monoswap.sol#540)
                - pools[_token].vusdDebt = uint112(_vusdDebt) (contracts/Monoswap.sol#541)
```

Reentrancy here is harmless since tokens are transferred from the caller to the contract.

```
Reentrancy in Monoswap.removeLiquidity(address,uint256,address,uint256,uint256) (contracts/Monoswap.sol#337-365):
        External calls:
        - _mintFee(pool.pid,pool.lastPoolValue,poolValue) (contracts/Monoswap.sol#346)
                - monoXPool.mint(feeTo,pid,devLiquidity) (contracts/Monoswap.sol#214)
        - vUSD.mint(to,vusdOut) (contracts/Monoswap.sol#351)
        - monoXPool.safeTransferERC20Token(_token,to,tokenOut) (contracts/Monoswap.sol#354)
        - burn(to,pool.pid,liquidityIn) (contracts/Monoswap.sol#356)
                - monoXPool.burn(account,id,amount) (contracts/Monoswap.sol#173)
        State variables written after the call(s):
        - _syncPoolInfo(_token,0,vusdOut) (contracts/Monoswap.sol#358)
                - pools[_token].tokenBalance = uint112(tokenReserve) (contracts/Monoswap.sol#300)
                - pools[_token].lastPoolValue = poolValue (contracts/Monoswap.sol#302)
                - pools[_token].vusdCredit = uint112(_vusdCredit) (contracts/Monoswap.sol#532)
                - pools[_token].vusdDebt = uint112(_vusdDebt) (contracts/Monoswap.sol#533)
                - pools[_token].vusdCredit = uint112(_vusdCredit) (contracts/Monoswap.sol#540)
                - pools[_token].vusdDebt = uint112(_vusdDebt) (contracts/Monoswap.sol#541)
```

Reentrancy here is a false positive since monoXPool is controlled by MonoX.

```
Reentrancy in Monoswap.swapIn(address,address,address,address,uint256) (contracts/Monoswap.sol#690-741):
        External calls:
        - IERC20(tokenIn).safeTransferFrom(from,address(monoXPool),amountIn) (contracts/Monoswap.sol#696)
        - IERC20(tokenIn).safeTransferFrom(from,address(monoXPool),amountIn) (contracts/Monoswap.sol#699)
        - vusdLocal.burn(address(monoXPool),amountIn) (contracts/Monoswap.sol#719)
        - monoXPool.safeTransferERC20Token(tokenOut,to,amountOut) (contracts/Monoswap.sol#731)
        State variables written after the call(s):
        - _updateTokenInfo(tokenOut,tokenOutPrice,tradeVusdValue.add(oneSideFeesInVusd),0,0) (contracts/Monoswap.sol#732)
                - pools[_token].tokenBalance = uint112(_balance) (contracts/Monoswap.sol#557)
                - pools[_token].price = uint112(_price) (contracts/Monoswap.sol#558)
                - pools[_token].vusdCredit = uint112(_vusdCredit) (contracts/Monoswap.sol#532)
                - pools[_token].vusdDebt = uint112(_vusdDebt) (contracts/Monoswap.sol#533)
                - pools[_token].vusdCredit = uint112(_vusdCredit) (contracts/Monoswap.sol#540)
                - pools[_token].vusdDebt = uint112(_vusdDebt) (contracts/Monoswap.sol#541)
        - _updateTokenInfo(tokenOut,tokenOutPrice,tradeVusdValue.add(oneSideFeesInVusd),0,amountOut) (contracts/Monoswap.sol#732)
                - pools[_token].tokenBalance = uint112(_balance) (contracts/Monoswap.sol#557)
                - pools[_token].price = uint112(_price) (contracts/Monoswap.sol#558)
                - pools[_token].vusdCredit = uint112(_vusdCredit) (contracts/Monoswap.sol#532)
                - pools[_token].vusdDebt = uint112(_vusdDebt) (contracts/Monoswap.sol#533)
                - pools[_token].vusdCredit = uint112(_vusdCredit) (contracts/Monoswap.sol#540)
                - pools[_token].vusdDebt = uint112(_vusdDebt) (contracts/Monoswap.sol#541)
Reentrancy in Monoswap.swapOut(address,address,address,address,uint256) (contracts/Monoswap.sol#745-795):
        External calls:
        - IERC20(tokenIn).safeTransferFrom(from,address(monoXPool),amountIn) (contracts/Monoswap.sol#754)
        - IERC20(tokenIn).safeTransferFrom(from,address(monoXPool),amountIn) (contracts/Monoswap.sol#757)
        State variables written after the call(s):
        - _updateTokenInfo(tokenIn,tokenInPrice,0,tradeVusdValue.add(oneSideFeesInVusd),0) (contracts/Monoswap.sol#775)
                - pools[_token].tokenBalance = uint112(_balance) (contracts/Monoswap.sol#557)
                - pools[_token].price = uint112(_price) (contracts/Monoswap.sol#558)
```

Reentrancy here is a false positive because all the swap functions are protected by locking modifiers.

```
INFO:Detectors:
Monoswap._updateVusdBalance(address,uint256,uint256) (contracts/Monoswap.sol#513-548) contains a tautology or contradiction:
        - require(bool,string)(_vusdCredit >= 0 && _vusdDebt == 0,Monoswap: unofficial pool cannot bear debt) (contracts/Monoswap.sol
```

Tautology is listed in this report as HAL-09.

```
MonoXPool.constructor(address)._WETH (contracts/MonoXPool.sol#19) lacks a zero-check on :
                - WETH = _WETH (contracts/MonoXPool.sol#20)
Monoswap.setFeeTo(address)._feeTo (contracts/Monoswap.sol#130) lacks a zero-check on :
                - feeTo = _feeTo (contracts/Monoswap.sol#131)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
```

Address validation is listed in this report as HAL-07.

FINDINGS & TECH DETAILS

```
INFO:Detectors:
mint(address,uint256,uint256) should be declared external:
        - MonoXPool.mint(address,uint256,uint256) (contracts/MonoXPool.sol#26-29)
burn(address,uint256,uint256) should be declared external:
        - MonoXPool.burn(address,uint256,uint256) (contracts/MonoXPool.sol#31-34)
initialize(MonoXPool,IvUSD) should be declared external:
        - Monoswap.initialize(MonoXPool,IvUSD) (contracts/Monoswap.sol#115-124)
updatePoolStatus(address,Monoswap.PoolStatus) should be declared external:
        - Monoswap.updatePoolStatus(address,Monoswap.PoolStatus) (contracts/Monoswap.sol#145-148)
updatePoolPrice(address,uint112) should be declared external:
        - Monoswap.updatePoolPrice(address,uint112) (contracts/Monoswap.sol#156-166)
listNewToken(address,uint112,uint256,uint256,address) should be declared external:
        - Monoswap.listNewToken(address,uint112,uint256,uint256,address) (contracts/Monoswap.sol#231-237)
balanceOf(address,uint256) should be declared external:
        - Monoswap.balanceOf(address,uint256) (contracts/Monoswap.sol#797-799)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
```

Function visibility is listed in this report as HAL-10.

MonoX Staking

```
INFO:Detectors:
MonoswapStaking.pendingMono(uint256,address) (contracts/MonoswapStaking.sol#192-233) performs a multiplication on the result of a division:
        -monoReward = monoReward.add(user.oldReward.mul(stakedAmount).div(user.amount).mul(1e12)) (contracts/MonoswapStaking.sol#221)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
MonoswapStaking.withdraw(uint256,uint256) (contracts/MonoswapStaking.sol#365-410) uses a dangerous strict equality:
        - user.amount == 0 (contracts/MonoswapStaking.sol#391)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
```

Multiplication on the result of division is listed in this report as HAL-09.

Strict equality is a false positive here since MonoX controls the user.amount value.

```
INFO:Detectors:
Reentrancy in MonoswapStaking.add(uint256,IERC1155,uint256,bool) (contracts/MonoswapStaking.sol#110-140):
        External calls:
        - massUpdatePools() (contracts/MonoswapStaking.sol#122)
                - mono.mint(address(this),monoReward.div(1e12)) (contracts/MonoswapStaking.sol#265)
        State variables written after the call(s):
        - poolInfo.push(PoolInfo(_lpToken,_lpTokenId,0,_allocPoint,lastRewardBlock,0,0,new address[](0),0,true)) (contracts/MonoswapStaking.sol#126-139)
        - totalAllocPoint = totalAllocPoint.add(_allocPoint) (contracts/MonoswapStaking.sol#125)
```

Reentrancy here is a false positive because all external calls are made to MonoX-controlled contracts.

```
INFO:Detectors:
MonoswapStaking.pendingMono(uint256,address) (contracts/MonoswapStaking.sol#192-233) compares to a boolean constant:
        -pool.bActive == true (contracts/MonoswapStaking.sol#202)
MonoswapStaking.migratePool(uint256,uint256) (contracts/MonoswapStaking.sol#284-323) compares to a boolean constant:
        -require(bool,string)(oldPool.bActive == false && newPool.bActive == true,migrate: wrong pools) (contracts/MonoswapStaking.sol#287)
MonoswapStaking.deposit(uint256,uint256) (contracts/MonoswapStaking.sol#326-362) compares to a boolean constant:
        -require(bool,string)(pool.bActive == true,deposit: stopped pool) (contracts/MonoswapStaking.sol#329)
MonoswapStaking.withdraw(uint256,uint256) (contracts/MonoswapStaking.sol#365-410) compares to a boolean constant:
        -pool.bActive == true (contracts/MonoswapStaking.sol#372)
MonoswapStaking.withdraw(uint256,uint256) (contracts/MonoswapStaking.sol#365-410) compares to a boolean constant:
        -pool.bActive == true (contracts/MonoswapStaking.sol#369)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality
```

The expressions reported here can be corrected so that variables are not unnecessarily compared to boolean constants.

Reference: Boolean Equality

```
INFO:Detectors:
setMinter(address) should be declared external:
        - MonoToken.setMinter(address) (contracts/MonoToken.sol#23-25)
mint(address,uint256) should be declared external:
        - MonoToken.mint(address,uint256) (contracts/MonoToken.sol#27-32)
initialize(MonoToken,uint256,uint256,uint256) should be declared external:
        - MonoswapStaking.initialize(MonoToken,uint256,uint256,uint256) (contracts/MonoswapStaking.sol#86-102)
add(uint256,IERC1155,uint256,bool) should be declared external:
        - MonoswapStaking.add(uint256,IERC1155,uint256,bool) (contracts/MonoswapStaking.sol#110-140)
set(uint256,uint256,bool) should be declared external:
        - MonoswapStaking.set(uint256,uint256,bool) (contracts/MonoswapStaking.sol#143-155)
stopPool(uint256) should be declared external:
        - MonoswapStaking.stopPool(uint256) (contracts/MonoswapStaking.sol#276-281)
migratePool(uint256,uint256) should be declared external:
        - MonoswapStaking.migratePool(uint256,uint256) (contracts/MonoswapStaking.sol#284-323)
deposit(uint256,uint256) should be declared external:
        - MonoswapStaking.deposit(uint256,uint256) (contracts/MonoswapStaking.sol#326-362)
withdraw(uint256,uint256) should be declared external:
        - MonoswapStaking.withdraw(uint256,uint256) (contracts/MonoswapStaking.sol#365-410)
emergencyWithdraw(uint256) should be declared external:
        - MonoswapStaking.emergencyWithdraw(uint256) (contracts/MonoswapStaking.sol#413-426)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
```

Function visibility is listed in this report as HAL-10.

3.14 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues, and to identify low-hanging fruit on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the testers machine and sent the compiled results to the analyzers to locate any vulnerabilities. In addition, security detections are only in scope.

Results:

Monoswap.sol

Report for contracts/Monoswap.sol
https://dashboard.mythx.io/#/console/analyses/78ff615c-9135-4260-9d3a-f960b1fd0261

Line	SWC Title	Severity	Short Description
25	(SWC-123) Requirement Violation	Low	Requirement violation.
30	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
31	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
32	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
33	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
149	(SWC-000) Unknown	Medium	Function could be marked as external.
179	(SWC-000) Unknown	Medium	Function could be marked as external.
181	(SWC-110) Assert Violation	Low	An assertion violation was triggered.
190	(SWC-000) Unknown	Medium	Function could be marked as external.
197	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
199	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
220	(SWC-000) Unknown	Medium	Function could be marked as external.
225	(SWC-000) Unknown	Medium	Function could be marked as external.
257	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
292	(SWC-000) Unknown	Medium	Function could be marked as external.
301	(SWC-000) Unknown	Medium	Function could be marked as external.
356	(SWC-107) Reentrancy	Low	Read of persistent state following external call.
409	(SWC-000) Unknown	Medium	Function could be marked as external.
485	(SWC-107) Reentrancy	Low	Read of persistent state following external call.
641	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
877	(SWC-000) Unknown	Medium	Function could be marked as external.

No new vulnerabilities were found by MythX.

MonoXPool.sol

Report for contracts/MonoXPool.sol
https://dashboard.mythx.io/#/console/analyses/bab51a84-c502-4a9b-a918-7b42c6c646fe

Line	SWC Title	Severity	Short Description
3	(SWC-103) Floating Pragma	Low	A floating pragma is set.
26	(SWC-000) Unknown	Medium	Function could be marked as external.
31	(SWC-000) Unknown	Medium	Function could be marked as external.

No new vulnerabilities were found by MythX.

VUSD.sol

No issues were found by MythX

THANK YOU FOR CHOOSING

HALBORN