Competitive Security Assessment

Shield M-Vault

Oct 18th, 2022

Secure3

secure3.io

Summary

M-Vault is an automated option selling vault that focuses on mainstream on-chain assets with decent liquidity, which earn yields through selling options to professional market makers.

This report has been prepared for the project to identify issues and vulnerabilities in the smart contract source code. A comprehensive examination with Static Analysis and Manual Review techniques has been performed by Secure3 team. Also, a group of KYC-and-NDA'ed experienced security experts have participated in the Secure3's Competitive Auditing as well to provide extra auditing coverage and scrutiny of the code.

The examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static scanner to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in three severity levels: Informational, Low, Medium, Critical. For each of the findings we have provided recommendation of a fix or mitigation for security and best practices.

Overview

Project Detail

Project Name	Shield M-Vault
Platform & Language	Ethereum, Solidity
Codebase	• repo - https://github.com/shielddeveloper/shield-vaults-v1 • audit commit - 7b69ff870be49d658d449ce5a96c8a200c711972 • final commit - f22005735cbe92c511b27d47cf70579a88ae26d2
Audit Methodology	• Competitive Auditing • Business Logic and Code Review • Privileged Roles Review • Static Analysis

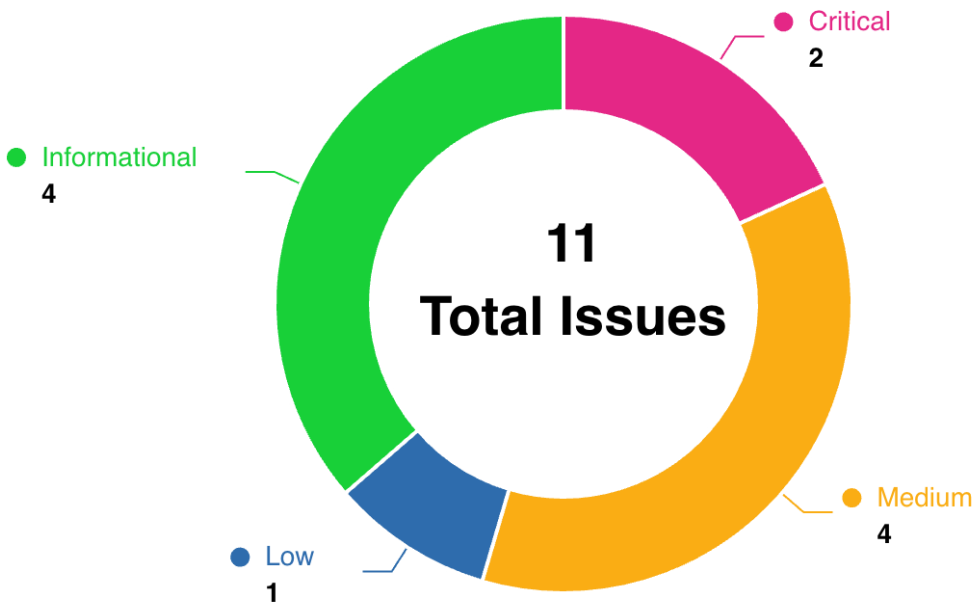
Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	2	0	1	1	0	0
Medium	4	0	2	2	0	0
Low	1	0	0	1	0	0
Informational	4	0	0	4	0	0

Audit Scope

File	Commit Hash
contracts/mvault/SSVault.sol	7b69ff870be49d658d449ce5a96c8a200c711972
contracts/mvault/MVaultManager.sol	7b69ff870be49d658d449ce5a96c8a200c711972
contracts/mvault/OracleManager.sol	7b69ff870be49d658d449ce5a96c8a200c711972
contracts/mvault/DEXv2Oracle.sol	7b69ff870be49d658d449ce5a96c8a200c711972
contracts/mvault/DEXv3Oracle.sol	7b69ff870be49d658d449ce5a96c8a200c711972
contracts/mvault/ChainlinkOracle.sol	7b69ff870be49d658d449ce5a96c8a200c711972
contracts/mvault/OracleFactory.sol	7b69ff870be49d658d449ce5a96c8a200c711972
contracts/mvault/Oracle.sol	7b69ff870be49d658d449ce5a96c8a200c711972

Code Assessment Findings



ID	Name	Category	Severity	Status	Contributor
SMV-1	`updateIndexPrice()` updating price in a wrong way, making it vulnerable to flash loan attacks	Oracle Manipulation	Critical	Acknowledged	iczc
SMV-2	`updateIndexPrice()` TWAP calculation is incorrect	Logical	Low	Fixed	bixia
SMV-3	Integer overflow risks when calculating token price	Integer Overflow and Underflow	Informational	Fixed	iczc
SMV-4	`consult()` should check if `secondsAgo` is valid	Logical	Informational	Fixed	bixia
SMV-5	`MVault` storage layout can be optimized to save gas	Gas Optimization	Informational	Fixed	bixia
SMV-6	`MVault` has reentrancy risk when `baseToken` is ERC777	Reentrancy	Critical	Fixed	w2ning, zzzix, Secure3
SMV-7	`settleOrders()` should check if `_users` is already settled	Logical	Medium	Fixed	bixia

SMV-8	Gas price is relatively stable in BSC	Logical	Medium	Acknowledged	iczc
SMV-9	Should check if `_token` is one of the `_aggregator` pair tokens	Logical	Medium	Fixed	bixia, iczc, zzzix
SMV-10	Use modifier instead of require statements	Code Style	Informational	Fixed	iczc
SMV-11	`getTokenPrice()` is insecure way to get prices from Uniswap v2 and v3	Oracle Manipulation	Medium	Acknowledged	bixia

SMV-1:`updateIndexPrice()` updating price in a wrong way, making it vulnerable to flash loan attacks

Category	Severity	Code Reference	Status	Contributor
Oracle Manipulation	Critical	contracts/mvault/DEXv2Oracle.sol#65	Acknowledged	iczc

Code

```
    65:     if (blockTimestampLast == 0) {
    66:         price0CumulativeLast = pair.price0CumulativeLast(); // fetch the current accumulated price value (1 / 0)
    67:         price1CumulativeLast = pair.price1CumulativeLast(); // fetch the current accumulated price value (0 / 1)
    68:
    69:         uint112 reserve0;
    70:         uint112 reserve1;
    71:         (reserve0, reserve1, blockTimestampLast) = pair.getReserves();
    72:
    73:         price0Average = FixedPoint.fraction(reserve1, reserve0);
    74:         price1Average = FixedPoint.fraction(reserve0, reserve1);
    75:
    76:         priceRouterCumulativeLast = block.timestamp;
    77:
    78:         if (router != address(0)) {
    79:             priceRouterAverage = getRouterPrice();
    80:         }
    81:     } else {
```

Description

iczc : DEXv2Oracle.sol is a price oracle build on Uniswap V2, and its `updateIndexPrice()` function has two branchings, specifically if blockTimestampLast is equal to 0(means never been updated) the priceAverage computed by consulting Uniswap pool reserve, otherwise using the time-weighted average prices (TWAPs) algorithm to compute the priceAverage.

The price calculated by reserve in branch 1 is actually the instantaneous price, an attacker can manipulate the price from the first-time `updateIndexPrice()` call by making a huge trade.

The following is the complete exploit concept:

1. Swap a huge amount of token0 to token1 in the liquidity pool which is used by the oracle
2. Calling the DEXv2Oracle's `updateIndexPrice()` function that has never been called
3. Use some function of a dApp to convince it to consult the oracle at the instant when the price has been manipulated to get profits
4. Execute another huge trade to swap token1 back to token0

Attacker can finish the flow in a single transaction by using a smart contracts, and it's possible to get initial funding with Flashloan as well, this means the attacker only loses fees and can’t get arbitrated.

Recommendation

iczc : 1. Acquire blockTimestampLast, price0CumulativeLast and price1CumulativeLast from Uniswap V2 pair and initialize these global variables in the constructor and `initIndexPrice()`, thus there is no need to handle the first-time consult special case in the `updateIndexPrice()` function.

```
    constructor(
        address _aggregator,
        address _router,
        uint256 _start
    ) ShieldOracle(_aggregator, _router) {
        IUniswapV2Pair p = IUniswapV2Pair(pair);
        token0 = p.token0();
        token1 = p.token1();
        price0CumulativeLast = p.price0CumulativeLast(); // fetch the current accumulated price
value (1 / 0)
        price1CumulativeLast = p.price1CumulativeLast(); // fetch the current accumulated price
value (0 / 1)
        startTime = _start;
        uint112 reserve0;
        uint112 reserve1;
        (reserve0, reserve1, blockTimestampLast) = p.getReserves();
        require(reserve0 != 0 && reserve1 != 0, "ShieldOracle: NO_RESERVES"); // ensure that there's
liquidity in the pair
    }
```

2. Build DEXv2Oracle's `updateIndexPrice()` directly into the critical calls of SSVault contracts, like `deposit()`.
3. Call the oracle frequently enough to update the latest cumulative price and save average price.

Client Response

This is a known behaviour. We believe the attacker does not have economical incentive to conduct the attack because attacker has to deposit token assets to the vault as preparation and when the price is manipulated and the price goes up, the option will be exercised which will cause the principal value lose. The recommendation will fix the issue, but it does not fit into the business feature, the vault only needs the TWAP price 30 min prior the settlement.

SMV-2:updateIndexPrice() TWAP calculation is incorrect

Category	Severity	Code Reference	Status	Contributor
Logical	Low	contracts/mvault/DEXv2Oracle.sol#L102-L112	Fixed	bixia

Code

```
101:     if (router != address(0)) {
102:         priceRouterAverage =
103:             (priceRouterAverage *
104:                 priceRouterCumulativeLast +
105:                 getRouterPrice() *
106:                 block.timestamp) /
107:             (priceRouterCumulativeLast + block.timestamp);
108:
109:         priceRouterCumulativeLast =
110:             priceRouterCumulativeLast +
111:             block.timestamp;
112:     }
```

Description

bixia : The way it calculate average router price is not consistent with the average price of TWAP. for the TWAP calculation method, it is

```
uint32 timeElapsed = blockTimestamp - blockTimestampLast;
price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
```

namely:

$$TWAP_{price0,i} = TWAP_{price0,i-1} + \frac{reserve_1}{reserve_0} * (T_i - T_{i-1})$$

$$TWAP_{price1,i} = TWAP_{price1,i-1} + \frac{reserve_0}{reserve_1} * (T_i - T_{i-1})$$

however, for the router price, the calculation method is:

```
priceRouterAverage = (
                    priceRouterAverage * priceRouterCumulativeLast
                        + getRouterPrice() * block.timestamp
                ) / (priceRouterCumulativeLast + block.timestamp);
priceRouterCumulativeLast = priceRouterCumulativeLast + block.timestamp;
```

namely:

$$\overline{price_{router,i}} = \frac{\overline{price_{router,i-1}} * T_{i-1} + price_{router,i} * T_i}{T_{i-1} + T_i}$$

$$\overline{price_{router,i}} = \overline{price_{router,i-1}} * \frac{T_{i-1}}{T_{i-1} + T_i} + price_{router,i} * \frac{T_i}{T_{i-1} + T_i}$$

as we can see, the way it calculate avg router price is not TWAP. it should be smaller than TWAP price.

Recommendation

bixia : use TWAP calculation method to calculate the router price.

```
uint timeElapsed = block.timestamp - blockTimestampLast;
priceRouterAverage = (
                    priceRouterAverage * priceRouterCumulativeLast
                        + getRouterPrice() * timeElapsed
                ) / (priceRouterCumulativeLast + timeElapsed);
priceRouterCumulativeLast = priceRouterCumulativeLast + timeElapsed;
```

Client Response

Fixed

SMV-3:Integer overflow risks when calculating token price

Category	Severity	Code Reference	Status	Contributor
Integer Overflow and Underflow	Informational	contracts/mvault/DEXv2Oracle.sol#195 contracts/mvault/DEXv3Oracle.sol#94 contracts/mvault/OracleManager.sol#319	Fixed	iczc

Code

```
// File: DEXv2Oracle.sol
195:    price = price * (10**(DECIMALS - decimalsOut));

// File: DEXv3Oracle.sol
94:    price = price * (10**(DECIMALS - decimalsOut));

// File: OracleManager.sol
319:    price = price * (10**(DECIMALS - decimalsOut));
```

Description

iczc : There is an integer overflow potential in the multiplication of price = price * (10**(DECIMALS - decimalsOut)); statement when price is very large and decimalsOut is less than 18.

Recommendation

iczc : Use Openzeppelin SafeMath for unit256 or upgrade solidity pragma to 0.8

Client Response

Fixed

SMV-4:`consult()` should check if `secondsAgo` is valid

Category	Severity	Code Reference	Status	Contributor
Logical	Informational	contracts/mvault/DEXv3Oracle.sol#L60-L63	Fixed	bixia

Code

```
    60:    function consult(uint32 secondsAgo) public view returns (uint256 price) {
```

Description

bixia : compared with the UniswapV3.OracleLibrary contract, the way it implement the consult function lacks one requirement, namely:

```
require(secondsAgo != 0, 'BP');
```

Recommendation

bixia : add the require sentence inside the consult function.

```
    function consult(uint32 secondsAgo) public view returns (uint256 price) {
        require(token == token0 || token == token1, "invalid token");
        require(secondsAgo != 0, 'BP');
        address tokenOut = token == token0 ? token1 : token0;
        ...
}
```

Client Response

Fixed

SMV-5:`MVault` storage layout can be optimized to save gas

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	Informational	contracts/mvault/MVault.sol#L20-L88 /code/contracts/mvault/MVaultManager.sol#L52-L57	Fixed	bixia

Code

```
    // File: MVault.sol
    20:    contract MVault is ReentrancyGuard, CommonType {
    21:    using SafeMath for uint256;
    22:    using EnumerableSet for EnumerableSet.AddressSet;
    23:
    24:    uint256 internal constant MULTIPLIER = 1e18;
    25:    uint256 internal constant PRECISION_LOSS = 10;
    // ...

    // File: MVaultManager.sol
    16:    contract MVaultManager is CommonType {
    17:    using SafeMath for uint256;
    18:
    19:    uint256 internal constant MULTIPLIER = 1e18;
    20:    uint256 internal constant SECONDS_IN_YEAR = 365 days;
    21:    uint256 internal constant meanPricePeriod = 30 minutes;
    // ...
```

Description

bixia : re-organize slots for contract `MVaultManager` and `MVault`

```
contract MVaultManager:
prev
    struct OracleInfo {
        bool valid;
        address oracle;
        address token1;
        address router;
        DexType version;
    }
after
    struct OracleInfo {
        bool valid;                     /// bytes1
        DexType version;          /// bytes1
        address oracle;             /// bytes20  => total occupy 1 slot
        address token1;
        address router;
    }
contract
```

Recommendation

bixia : basically, you can modify those global params with `immutable`, which set inside the constructor and will not be modified, to save gas. because `sload` is usually much expensive than the normal `codecopy`

```
contract MVault:
    uint256 internal constant MULTIPLIER = 1e18;
    uint256 internal constant PRECISION_LOSS = 10;
    uint256 internal constant SECONDS_IN_YEAR = 365 days;
    uint256 internal constant BASE_GAS_AMOUNT = 21000;
    address private immutable sldToken;
    address private immutable factory;
    address public immutable publisher;
    address private immutable sldAggregator;

    IBEP20 public baseToken;
    Oracle public oracle;
    bool public terminated;

    RoundInfo[] public roundInfo;
    uint256 public totalActiveDeposit;
    uint256 public totalPendingDeposit;
    uint256 public totalPendingWithdraw;
    uint256 public totalWithdrawable;
    uint256 public baseMargin;
    uint256 public sldMargin;
    uint256 public minSLDMargin;
    uint256 public minPeriod;
    uint256 public updatePricePeriod;
    uint256 public quotePeriod;
    uint256 public feeRate;
    uint256 private gasPrice;

    // Chainlink price feeder
    IAggregatorV3 private GasAggregator;
    EnumerableSet.AddressSet private users;
    mapping(address => UserInfo) public userInfo;
    mapping(uint256 => uint256) private shouldSettled;
```

Client Response

Fixed

SMV-6:`MVault` has reentrancy risk when `baseToken` is ERC777

Category	Severity	Code Reference	Status	Contributor
Reentrancy	Critical	contracts/mvault/MVault.sol#L407 contracts/mvault/MVault.sol#L416-426 contracts/mvault/MVault.sol#L416	Fixed	w2ning, zzzix, Secure3

Code

```
416:     TransferHelper.safeTransfer(
417:         address(baseToken),
418:         msg.sender,
419:         withdrawable
420:     );
421:
422:     user.activeDeposit = 0;
423:     user.pendingDeposit = 0;
424:     user.pendingWithdraw = 0;
425:     user.withdrawable = 0;
426:     users.remove(msg.sender);
```

Description

w2ning : Reentrancy of MVault contract emergencyWithdraw function, this could result in lose of fund because the value is deducted after ...
zzzix : When the `baseToken` is a ERC777 token and registered for a callback function, malicious code can call the function again and because the state change is after the call, it can withdraw more than the user has.
Secure3 : The emergencyWithdraw() function has reentrancy issue when the token is ERC777 token due to the hook callback function. It can result in lose of fund because the value is only updated after the transfer call.

Recommendation

w2ning : use the Checks-Effects-Interactions best practice and make all state changes before calling external contracts. Also, consider using function modifiers such as Reentrancy Guard to prevent re-entrancy from contract level.

```
    // Remove user first
    user.activeDeposit = 0;
    user.pendingDeposit = 0;
    user.pendingWithdraw = 0;
    user.withdrawable = 0;
    users.remove(msg.sender);

    // Then transfer
    TransferHelper.safeTransfer(
        address(baseToken),
        msg.sender,
        withdrawable
    );
```

zzzix : consider use https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/ReentrancyGuard.sol modifier to prevent the call to reenter.
Secure3 : make state changes before calling `safeTransfer()` or use `nonReentrant` modifier to guard the function.

Client Response

Fixed

SMV-7:`settleOrders()` should check if `_users` is already settled

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	contracts/mvault/MVault.sol#L661-L687	Fixed	bixia

Code

```
661:     function settleOrder(
662:         address _user,
663:         bool _isStrike,
664:         uint256 _APY,
665:         uint256 _strikePrice,
666:         uint256 _settledPrice,
667:         uint256 _startTime,
668:         uint256 _endTime
669:     ) internal returns (uint256 fee) {
670:         UserInfo storage user = userInfo[_user];
671: 
672:         if (getLatestRoundId() == 0) {
673:             totalActiveDeposit = totalActiveDeposit.add(user.pendingDeposit);
674:             totalPendingDeposit = totalPendingDeposit.sub(user.pendingDeposit);
675:             totalWithdrawable = totalWithdrawable.add(user.pendingWithdraw);
676:             totalPendingWithdraw = totalPendingWithdraw.sub(
677:                 user.pendingWithdraw
678:             );
679:             user.activeDeposit = user.activeDeposit.add(user.pendingDeposit);
680:             user.pendingDeposit = 0;
681:             user.withdrawable = user.withdrawable.add(user.pendingWithdraw);
682:             user.pendingWithdraw = 0;
683: 
684:             return 0;
685:         }
686: 
687:         require(user.settledRound != getLatestRoundId(), "already settled");
```

Description

bixia : `settleOrders` function failed to check the input params, `address[] memory users`. which can be the same user that repeat multiple times. By doing this, the single user can occupy other users' chance to get settled for the round id = 0. you may refer to the following poc:

```
    function deposit() public {
        address alice = address(0x01);
        address bob = address(0x02);

        mvaultManager.createMVault(
            SLD,
            SLD_USDT,
            // evilPair,
            0.1 ether,
            0.26 ether,
            100000.0 ether,
            1 ether,
            block.timestamp + 30 minutes
        );
        address[] memory vaults = mvaultManager.getVaults(0, 1);
        address vault = vaults[0];
        address oracle = address(MVault(vault).oracle());

        vm.startPrank(alice);
        ERC20Like(SLD).approve(vault, type(uint256).max);
        MVault(vault).deposit(1 ether);
        vm.stopPrank();
        vm.startPrank(bob);
        ERC20Like(SLD).approve(vault, type(uint256).max);
        MVault(vault).deposit(1 ether);
        vm.stopPrank();
        ERC20Like(SLD).approve(vault, type(uint256).max);
        MVault(vault).deposit(1 ether);

        (, uint256 pendingDeposit,,,) = MVault(vault).userInfo(address(this));
        require(pendingDeposit == 1 ether);

        (, pendingDeposit,,,) = MVault(vault).userInfo(alice);
        require(pendingDeposit == 1 ether);

        (, pendingDeposit,,,) = MVault(vault).userInfo(bob);
        require(pendingDeposit == 1 ether);

        // MVault(vault).updatePrice();
        vm.warp(block.timestamp + 1 hours);
        require(MVault(vault).getLatestRoundId() == 0, "must be round id = 0");
        address[] memory users = new address[](3);
        users[0] = address(this);
        users[1] = address(this);
        users[2] = address(this);

        MVault(vault).settleOrders(users);
        (,,,,,,,,, bool settled) = MVault(vault).getLatestRoundInfo();
```

```
        require(settled, "must be");
    }
```

Recommendation

bixia : move the `require(user.settledRound != getLatestRoundId(), "already settled");` at the beginning of function `settleOrder`

```
    function settleOrder(
        address _user,
        bool _isStrike,
        uint256 _APY,
        uint256 _strikePrice,
        uint256 _settledPrice,
        uint256 _startTime,
        uint256 _endTime
    ) internal returns (uint256 fee) {
        UserInfo storage user = userInfo[_user];
        require(user.settledRound != getLatestRoundId(), "already settled");
        ...
        if (getLatestRoundId() == 0) {
            ...
            user.settledRound = getLatestRoundId();
        }
}
```

Client Response

Fixed with another implementation

SMV-8:Gas price is relatively stable in BSC

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	contracts/mvault/MVault.sol#761	Acknowledged	iczc

Code

```
761:     function sendSettleRewards(uint256 _gasUsed) internal {
762:         uint256 gasFeeUsed = _gasUsed
763:             .add(BASE_GAS_AMOUNT)
764:             .mul(gasPrice)
765:             .mul(getCoinPrice())
766:             .div(MULTIPLIER);
767:
768:         uint256 rewards = gasFeeUsed.mul(getSLDPrice()).div(MULTIPLIER);
769:
770:         if (rewards > 0) {
771:             sldMargin = sldMargin.sub(rewards);
772:             TransferHelper.safeTransfer(address(sldToken), msg.sender, rewards);
773:         }
774:     }
```

Description

iczc : MVault allows anyone to settle orders, and the settler receives SLD token subsidy of the same value according to the gas fee cost. Since the gas price for calculating the gas fee is fixed, when the current network gas price is less than the fixed gas price value, users can arbitrage by continuously settling orders to get low price SLD token.

Recommendation

iczc : Use the oracle to get the current network gas price.

Client Response

The SLD token price will be controlled by the Shield project owned address. This address will later be controlled by the DAO governance contract and the whole community and token holders.

SMV-9:Should check if `_token` is one of the `_aggregator` pair tokens

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	contracts/mvault/MVaultManager.sol#L95-L103 contracts/mvault/OracleManager.sol#182	Fixed	bixia, iczc, zzzix

Code

```
// File: MVaultManager.sol
95:       function createMVault(
96:           address _token,
97:           address _aggregator,
98:           uint256 _strikePrice,
99:           uint256 _APY,
100:          uint256 _maxVolume,
101:          uint256 _minDeposit,
102:          uint256 _endTime
103:      ) public {

// File: OracleManager.sol
182:      function checkAggregator(address _token, address _aggregator)
183:          public
184:          view
185:          returns (
186:              bool valid,
187:              address token0,
188:              address token1,
189:              DexType version,
190:              address router
191:          )
192:      {
```

Description

bixia : inside the MVaultManager.sol, the function `createMVault` failed to check the relationship between `token` and `aggregator`. which means that we can pass SLD as the token, but pass EvilToken_USDT pair as the aggreator into the `createVault` function. the main drawback here is that the `token` price can be manipulated by the fake aggregator pair. Hence, the vault owner can set super high yield APY to attract normal users to deposit in, but the vault owner can manipulate the oracle price to determine whether it should be strike or not. you may refer to the following POC code:

```
    function createMVault() public {
        evilToken = address(new EvilToken());
        address evilPair = address(factoryV2Like(factoryV2).createPair(USDT, address(evilToken)));
        ERC20Like(USDT).transfer(evilPair, 1 wei);
        ERC20Like(evilToken).transfer(evilPair, 1000 ether);
        PairLike(evilPair).sync();
        require(PairLike(evilPair).token1() == USDT, "must be");

        mvaultManager.createMVault(
            SLD,
            // SLD_USDT,
            evilPair,
            0.1 ether,
            0.26 ether,
            100000.0 ether,
            1000 ether,
            block.timestamp + 7 days
        );
        address[] memory vaults = mvaultManager.getVaults(0,1);
        address vault = vaults[0];
        address oracle = address(MVault(vault).oracle());
        require(DEXv2Oracle(oracle).token0() == evilToken , "not ok");
        require(DEXv2Oracle(oracle).token1() == USDT, "");
    }
```

iczc : If the `checkAggregator()` function is called with the _token parameter not matching _aggregator, meaning the _token does not belong to this _aggregator pair, the check still will execute successfully. This causes the mismatched vault not being able to query the correct price.

Recommendation

bixia : add sanity check between the `token` and `aggregator` inside the `oracleManager.checkAggregator` function. like the below:

```
    function checkAggregator(address _token, address _aggregator)
        public
        view
        returns (bool valid, address token0, address token1, DexType version, address router)
    {
     ...
        token0 = pair.token0();
        token1 = pair.token1();
        require(_token == token0 || _token == token1, "sanity check failed");
        if (token1 == _token) {
            token1 = token0;
            token0 = _token;
        }

      ...
     }
```

iczc : Check token is required in the aggregator pair.

```
require(_token == pair.token0() || _token == pair.token1(), "invalid token");
```

Client Response

Fixed

SMV-10:Use modifier instead of require statements

Category	Severity	Code Reference	Status	Contributor
Code Style	Informational	contracts/mvault/OracleManager.sol#122,145,167 contracts/mvault/OracleManager.sol#62,80,96,109 contracts/mvault/MVaultManager.sol#207,270,314,219,232,242,249,256,263,277,284,291,298,307	Fixed	iczc

Code

```
// File: OracleManager.sol
122:     require(msg.sender == keeper, "not keeper");
145:     require(msg.sender == keeper, "not keeper");
167:     require(msg.sender == keeper, "not keeper");

62:     require(msg.sender == governance, "not governance");
80:     require(msg.sender == governance, "not governance");
96:     require(msg.sender == governance, "not governance");
109:     require(msg.sender == governance, "not governance");

// File: MVaultManager.sol
207:     require(governance == msg.sender);
270:     require(governance == msg.sender);
314:     require(governance == msg.sender);

219:     require(governance == msg.sender, "not governance");
232:     require(governance == msg.sender, "not governance");
242:     require(governance == msg.sender, "not governance");
249:     require(governance == msg.sender, "not governance");
256:     require(governance == msg.sender, "not governance");
263:     require(governance == msg.sender, "not governance");
277:     require(governance == msg.sender, "not governance");
284:     require(governance == msg.sender, "not governance");
291:     require(governance == msg.sender, "not governance");
298:     require(governance == msg.sender, "not governance");
307:     require(governance == msg.sender, "not governance");
```

Description

iczc : There are 3 duplicate `require(msg.sender == keeper, "not keeper")` codes in OracleManager.sol to check access.
iczc : There are 4 duplicate `require(msg.sender == governance, "not governance")` codes in OracleManager.sol to check access.
iczc : There are 14 duplicate `require(governance == msg.sender, "not governance")` or `require(governance == msg.sender)`codes in OracleManager.sol to check access.

Recommendation

iczc : Define a modifier for these duplicate authentication logics and add the modifier to the functions which require checking access.

```
modifier onlyKeeper() {
    require(msg.sender == keeper, "not keeper");
    _;
}
```

iczc : Define a modifier for these duplicate authentication logics and add the modifier to the functions which require checking access.

```
modifier onlyGovernance() {
    require(msg.sender == governance, "not governance")
    _;
}
```

iczc : Define a modifier for these duplicate authentication logics and add the modifier to the functions which require checking access.

```
modifier onlyGovernance() {
    require(msg.sender == governance, "not governance")
    _;
}
```

Client Response

Fixed

SMV-11:`getTokenPrice()` is insecure way to get prices from Uniswap v2 and v3

Category	Severity	Code Reference	Status	Contributor
Oracle Manipulation	Medium	contracts/mvault/OracleManager.sol#L292-L301	Acknowledged	bixia

Code

```
292:     if (pair.factory() == v2Factory) {
293:         uint256 reserve0 = IBEP20(token0).balanceOf(_aggregator);
294:         uint256 reserve1 = IBEP20(token1).balanceOf(_aggregator);
295:
296:         price = (reserve1 * 10**decimalsIn) / reserve0;
297:     } else if (pair.factory() == v3Factory) {
298:         IUniswapV3Pool pool = IUniswapV3Pool(_aggregator);
299:         (uint160 sqrtPriceX96, , , , , , ) = pool.slot0();
```

Description

bixia : inside the OracleManager.sol, the function `getTokenPrice` should not use spot price both for the uniV2 like aggregator and uniV3 like aggregator. for the uniV2 like, it use the following formula to calculate the price:

$$price_{v2} = \frac{balance_{token1}}{balance_{token0}}$$

for the uniV3 like, it use the spot price stored inside the slot0. which is also easily get manipulated by swapping.

```
(uint160 sqrtPriceX96,,,,,,) = pool.slot0();
```

for the uniV2 like spot price, we can easily manipulate the price by transferring token1 into the pair contract, which increase the $balance_{token1}$, to drive the price up. for the uniV3 like spot price, we can easily manipulate it by swapping token0 for token1, which will move the slot0.sqrtPriceX96 far away from its true price, to drive the price up.

Recommendation

bixia : for the UniV2 like pool, you may implement the TWAP way to retrieve price. or get the price from `DEXv2Oracle.getIndexPrice()` for the uniV3 like pool, you should not use the slot0 for the price reference. you may use the `consult()` function inside the DEXv3Oracle.

Client Response

This function is only used for front end dApp query purpose, hence the actual impact of the flash loan attack is limited.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.