Finance.Vote - BasicPoolFactory

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **June 14th-June 22th, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	06/14/2021	Gabi Urrutia
0.2	Document Edits	06/18/2021	Gokberk Gulgun
1.0	Final Version	06/22/2021	Gabi Urrutia
1.1	Remediation Plan	06/25/2021	Gokberk Gulgun

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
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Finance.Vote engaged Halborn to conduct a security assessment on their Smart contracts beginning on June 14th, 2021 and ending June 22th, 2021. The security assessment was scoped to the smart contract BasicPoolFactory.sol. An audit of the security risk and implications regarding the changes introduced by the development team at Finance.Vote prior to its production release shortly following the assessments deadline.

1.2 AUDIT SUMMARY

The team at Halborn was provided a week timeframe for the engagement and assigned two full time security engineers to audit the security of the smart contract. The security engineers are blockchain and smart contract security experts, with experience in advanced penetration testing, smart contract hacking, and have a deep knowledge in 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 smart contracts 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 read 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.
- Scanning of solidity files for vulnerabilities, security hotspots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions.(Slither)
- Testnet deployment (Truffle,Ganache)

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:

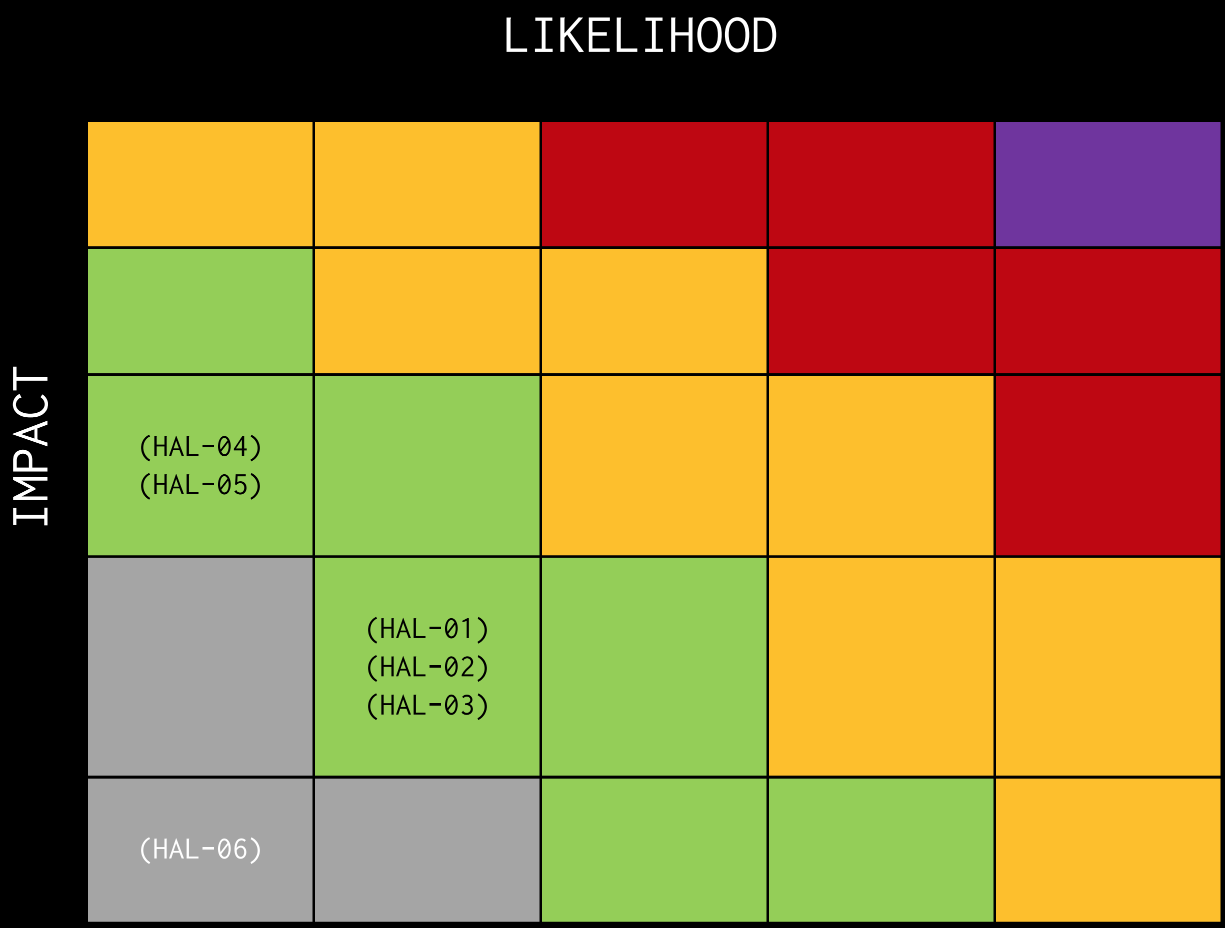
BasicPoolFactory.sol - Commit 7e1f247d7640edfe4bf68140328dd087c95c4700

BasicPoolFactory.sol - Fixed Commit ID 9433667973e86ebd76f3d3fe7d996086b73c2c0e

OUT-OF-SCOPE:
Other smart contracts in the repository, external libraries and economics attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	5	1



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - MISSING RE-ENTRANCY PROTECTION	Low	SOLVED: 06/25/2021
HAL02 - MISSING ADDRESS VALIDATION	Low	RISK ACCEPTED: 06/25/2021
HAL03 - MISSING EVENT HANDLER	Low	SOLVED: 06/25/2021
HAL04 - USE OF BLOCK.TIMESTAMP	Low	RISK ACCEPTED: 06/25/2021
HAL05 - IGNORED RETURN VALUES	Low	SOLVED: 06/25/2021
HAL06 - POSSIBLE MISUSE OF PUBLIC FUNCTIONS	Informational	SOLVED: 06/25/2021

FINDINGS & TECH DETAILS

3.1 (HAL-01) MISSING RE-ENTRANCY PROTECTION - LOW

Description:

Calling external contracts is dangerous if some functions and variables are called after the external call. An attacker could use a malicious contract to perform a recursive call before calling function and take over the control flow.

Code Location:

BasicPoolFactory.sol Line #~112

Listing 1: BasicPoolFactory.sol (Lines 112)

```
103     function deposit(uint poolId, uint amount) external {
104         Pool storage pool = pools[poolId];
105         require(pool.id == poolId, 'Uninitialized pool');
106         require(block.timestamp > pool.startTime, 'Cannot deposit
                before pool start');
107         require(block.timestamp < pool.endTime, 'Cannot deposit
                after pool ends');
108         require(pool.totalDeposits < pool.maximumDeposit, 'Maximum
                 deposit already reached');
109         if (pool.totalDeposits.plus(amount) > pool.maximumDeposit)
                 {
110             amount = pool.maximumDeposit.minus(pool.totalDeposits)
                    ;
111         }
112         IERC20(pool.depositToken).transferFrom(msg.sender, address
                (this), amount);
113         pool.totalDeposits = pool.totalDeposits.plus(amount);
114         pool.numReceipts = pool.numReceipts.plus(1);
115
116         Receipt storage receipt = pool.receipts[pool.numReceipts];
117         receipt.id = pool.numReceipts;
118         receipt.amountDeposited = amount;
119         receipt.timeDeposited = block.timestamp;
120         receipt.owner = msg.sender;
```

```

        emit DepositOccurred(poolId, pool.numReceipts, msg.sender)
            ;
    }

```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

As possible, external calls should be at the end of the function in order to to avoiding an attacker take over the control flow. If not possible, deploy some locking mechanism, like the commonly known ReentrancyGuard instead. Make sure that any pair of code paths that have a possible read/write conflict for a variable will be “reentrancy guarded”.

Remediation Plan:

SOLVED: Finance.Vote Team moved an external call to the end of the function.

Listing 2: BasicPoolFactory.sol (Lines 123)

```
105 function deposit(uint poolId, uint amount) external {
106         Pool storage pool = pools[poolId];
107         require(pool.id == poolId, 'Uninitialized pool');
108         require(block.timestamp > pool.startTime, 'Cannot deposit
                before pool start');
109         require(block.timestamp < pool.endTime, 'Cannot deposit
                after pool ends');
110         require(pool.totalDeposits < pool.maximumDeposit, 'Maximum
                 deposit already reached');
111         if (pool.totalDeposits.plus(amount) > pool.maximumDeposit)
                 {
112             amount = pool.maximumDeposit.minus(pool.totalDeposits)
                    ;
```

```
        }
        pool.totalDeposits = pool.totalDeposits.plus(amount);
        pool.numReceipts = pool.numReceipts.plus(1);

        Receipt storage receipt = pool.receipts[pool.numReceipts];
        receipt.id = pool.numReceipts;
        receipt.amountDeposited = amount;
        receipt.timeDeposited = block.timestamp;
        receipt.owner = msg.sender;

        bool success = IERC20(pool.depositToken).transferFrom(msg.sender, address(this), amount);
        require(success, 'Token transfer failed');

        emit DepositOccurred(poolId, pool.numReceipts, msg.sender);
    }

```

3.2 (HAL-02) MISSING ADDRESS VALIDATION - LOW

Description:

BasicPoolFactory.sol contract is missing a safety check inside their constructors and multiple functions. Setters of address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever.

Code Location:

BasicPoolFactory.sol Line #~48

Listing 3: BasicPoolFactory.sol (Lines)

```
48     constructor (address mgmt) {
49         management = mgmt;
50     }
51
52     // change the management key
53     function setManagement(address newMgmt) public managementOnly
           {
54         address oldMgmt = management;
55         management = newMgmt;
56         emit ManagementUpdated(oldMgmt, newMgmt);
57     }
```

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.

For example:

Listing 4: Modifier.sol (Lines 2,3,4)

```
modifier validAddress(address addr) {
    require(addr != address(0), "Address cannot be 0x0");
    require(addr != address(this), "Address cannot be contract");
    _;
}
```

Remediation Plan:

RISK ACCEPTED: Finance.Vote Team decided to continue without address validation.

3.3 (HAL-03) MISSING EVENT HANDLER - LOW

Description:

In the BasicPoolFactory.sol contract the function does not emit event after the progress. Events are a method of informing the transaction initiator about the actions taken by the called function. It logs its emitted parameters in a specific log history, which can be accessed outside of the contract using some filter parameters.

Code Location:

BasicPoolFactory.sol Line #~48

Listing 5: BasicPoolFactory.sol (Lines)

```
    function addPool (
        uint startTime,
        uint maxDeposit,
        uint[] memory rewardsPerSecondPerToken,
        uint programLengthDays,
        address depositTokenAddress,
        address[] memory rewardTokenAddresses
    ) public managementOnly {
        numPools = numPools.plus(1);
        Pool storage pool = pools[numPools];
        pool.id = numPools;
        pool.rewardsPerSecondPerToken = rewardsPerSecondPerToken;
        pool.startTime = startTime > block.timestamp ? startTime :
            block.timestamp;
        pool.endTime = startTime.plus(programLengthDays * 1 days);
        pool.depositToken = depositTokenAddress;
        require(rewardsPerSecondPerToken.length ==
            rewardTokenAddresses.length, 'Rewards and reward token
            arrays must be same length');

        for (uint i = 0; i < rewardTokenAddresses.length; i++) {
            pool.rewardTokens.push(rewardTokenAddresses[i]);
            pool.rewardsClaimed.push(0);
```

```
        }

        pool.maximumDeposit = maxDeposit;
    }
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Consider as much as possible declaring events at the end of function. Events can be used to detect the end of the operation.

Remediation Plan:

SOLVED: Finance.Vote Team added event at the end of the function.

Listing 6: BasicPoolFactory.sol (Lines 83)

```
    function addPool (
        uint startTime,
        uint maxDeposit,
        uint[] memory rewardsPerSecondPerToken,
        uint programLengthDays,
        address depositTokenAddress,
        address[] memory rewardTokenAddresses
    ) external managementOnly {
        numPools = numPools.plus(1);
        Pool storage pool = pools[numPools];
        pool.id = numPools;
        pool.rewardsPerSecondPerToken = rewardsPerSecondPerToken;
        pool.startTime = startTime > block.timestamp ? startTime :
            block.timestamp;
        pool.endTime = startTime.plus(programLengthDays * 1 days);
        pool.depositToken = depositTokenAddress;
        require(rewardsPerSecondPerToken.length ==
            rewardTokenAddresses.length, 'Rewards and reward token
```

```
            arrays must be same length');

        for (uint i = 0; i < rewardTokenAddresses.length; i++) {
            pool.rewardTokens.push(rewardTokenAddresses[i]);
            pool.rewardsClaimed.push(0);
        }

        pool.maximumDeposit = maxDeposit;
        emit PoolAdded(pool.id);
    }

```

3.4 (HAL-04) USE OF BLOCK.TIMESTAMP - LOW

Description:

The global variable block.timestamp does not necessarily hold the current time, and may not be accurate. Miners can influence the value of block .timestamp to perform Maximal Extractable Value (MEV) attacks. There is no guarantee that the value is correct, only that it is higher than the previous block's timestamp.

Code Location:

Listing 7: BasicPoolFactory.sol (Lines 149)

```
148 function addPool (
149         uint startTime,
150         uint maxDeposit,
151         uint[] memory rewardsPerSecondPerToken,
152         uint programLengthDays,
153         address depositTokenAddress,
154         address[] memory rewardTokenAddresses
155     ) public managementOnly {
156         numPools = numPools.plus(1);
157         Pool storage pool = pools[numPools];
158         pool.id = numPools;
159         pool.rewardsPerSecondPerToken = rewardsPerSecondPerToken;
160         pool.startTime = startTime > block.timestamp ? startTime :
                block.timestamp;
161         pool.endTime = startTime.plus(programLengthDays * 1 days);
162         pool.depositToken = depositTokenAddress;
163         require(rewardsPerSecondPerToken.length ==
               rewardTokenAddresses.length, 'Rewards and reward token
               arrays must be same length');
164
165         for (uint i = 0; i < rewardTokenAddresses.length; i++) {
166             pool.rewardTokens.push(rewardTokenAddresses[i]);
167             pool.rewardsClaimed.push(0);
168         }
169
```

```
170         pool.maximumDeposit = maxDeposit;
171     }
172
```

Listing 8: BasicPoolFactory.sol (Lines 54)

```
53 function getRewards(uint poolId, uint receiptId) public view returns (uint[] memory) {
54         Pool storage pool = pools[poolId];
55         Receipt memory receipt = pool.receipts[receiptId];
56         require(pool.id == poolId, 'Uninitialized pool');
57         require(receipt.id == receiptId, 'Uninitialized receipt');
58         uint nowish = block.timestamp;
59         if (nowish > pool.endTime) {
60             nowish = pool.endTime;
61         }
62
63         uint secondsDiff = nowish.minus(receipt.timeDeposited);
64         uint[] memory rewardsLocal = new uint[](pool.rewardsPerSecondPerToken.length);
65         for (uint i = 0; i < pool.rewardsPerSecondPerToken.length; i++) {
66             rewardsLocal[i] = (secondsDiff.times(pool.rewardsPerSecondPerToken[i]).times(receipt.amountDeposited)) / 1e18;
67         }
68
69         return rewardsLocal;
70 }
```

Listing 9: BasicPoolFactory.sol (Lines 25)

```
24 function withdrawExcessRewards(uint poolId) external {
25         Pool storage pool = pools[poolId];
26         require(pool.id == poolId, 'Uninitialized pool');
27         require(pool.totalDeposits == 0, 'Cannot withdraw until all deposits are withdrawn');
28         require(block.timestamp > pool.endTime, 'Contract must reach maturity');
29
30         for (uint i = 0; i < pool.rewardTokens.length; i++) {
31             IERC20 rewardToken = IERC20(pool.rewardTokens[i]);
32             uint rewards = rewardToken.balanceOf(address(this));
```

```
            rewardToken.transfer(management, rewards);
        }
        IERC20 depositToken = IERC20(pool.depositToken);
        depositToken.transfer(management, depositToken.balanceOf(
            address(this)));
        emit ExcessRewardsWithdrawn(poolId);
    }

```

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:

RISK ACCEPTED: Finance.Vote Team decided to continue with block.timestamp

3.5 (HAL-05) IGNORED RETURN VALUES - LOW

Description:

The return value of an external call is not stored in a local or state variable. In the BasicPoolFactory contract, there are a few instances where the multiple methods are called and the return value (bool) is ignored.

BasicPoolFactory.sol Line #~103,125,147

Listing 10: BasicPoolFactory.sol (Lines 112)

```
103     function deposit(uint poolId, uint amount) external {
104         Pool storage pool = pools[poolId];
105         require(pool.id == poolId, 'Uninitialized pool');
106         require(block.timestamp > pool.startTime, 'Cannot deposit before pool start');
107         require(block.timestamp < pool.endTime, 'Cannot deposit after pool ends');
108         require(pool.totalDeposits < pool.maximumDeposit, 'Maximum deposit already reached');
109         if (pool.totalDeposits.plus(amount) > pool.maximumDeposit) {
110             amount = pool.maximumDeposit.minus(pool.totalDeposits);
111         }
112         IERC20(pool.depositToken).transferFrom(msg.sender, address(this), amount);
113         pool.totalDeposits = pool.totalDeposits.plus(amount);
114         pool.numReceipts = pool.numReceipts.plus(1);
115 
116         Receipt storage receipt = pool.receipts[pool.numReceipts];
117         receipt.id = pool.numReceipts;
118         receipt.amountDeposited = amount;
119         receipt.timeDeposited = block.timestamp;
120         receipt.owner = msg.sender;
121 
122         emit DepositOccurred(poolId, pool.numReceipts, msg.sender);
```

```
123     }
124
```

Listing 11: BasicPoolFactory.sol (Lines 154,158)

```
147     function withdrawExcessRewards(uint poolId) external {
148         Pool storage pool = pools[poolId];
149         require(pool.id == poolId, 'Uninitialized pool');
150         require(pool.totalDeposits == 0, 'Cannot withdraw until
                all deposits are withdrawn');
151         require(block.timestamp > pool.endTime, 'Contract must
                reach maturity');
152
153         for (uint i = 0; i < pool.rewardTokens.length; i++) {
154             IERC20 rewardToken = IERC20(pool.rewardTokens[i]);
155             uint rewards = rewardToken.balanceOf(address(this));
156             rewardToken.transfer(management, rewards);
157         }
158         IERC20 depositToken = IERC20(pool.depositToken);
159         depositToken.transfer(management, depositToken.balanceOf(
                address(this)));
160         emit ExcessRewardsWithdrawn(poolId);
161     }
162
163
```

Listing 12: BasicPoolFactory.sol (Lines 141,143)

```
125     function withdraw(uint poolId, uint receiptId) external {
126         Pool storage pool = pools[poolId];
127         require(pool.id == poolId, 'Uninitialized pool');
128         Receipt storage receipt = pool.receipts[receiptId];
129         require(receipt.id == receiptId, 'Can only withdraw real
                receipts');
130         require(receipt.owner == msg.sender || block.timestamp >
                pool.endTime, 'Can only withdraw your own deposit');
131         require(receipt.timeWithdrawn == 0, 'Can only withdraw
                once per receipt');
132
133         // close re-entry gate
134         receipt.timeWithdrawn = block.timestamp;
135
136         uint[] memory rewards = getRewards(poolId, receiptId);
```

```
        pool.totalDeposits = pool.totalDeposits.minus(receipt.
            amountDeposited);

        for (uint i = 0; i < rewards.length; i++) {
            pool.rewardsClaimed[i] = pool.rewardsClaimed[i].plus(
                rewards[i]);
            IERC20(pool.rewardTokens[i]).transfer(receipt.owner,
                rewards[i]);
        }
        IERC20(pool.depositToken).transfer(receipt.owner, receipt.
            amountDeposited);
        emit WithdrawalOccurred(poolId, receiptId, receipt.owner);
    }

```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Add a return value check to avoid an unexpected crash of the contract. Return value checks provide better exception handling.

Remediation Plan:

SOLVED: Finance.Vote Team checked return values on the external calls.

Listing 13: BasicPoolFactory.sol (Lines 123)

```
    function deposit(uint poolId, uint amount) external {
        Pool storage pool = pools[poolId];
        require(pool.id == poolId, 'Uninitialized pool');
        require(block.timestamp > pool.startTime, 'Cannot deposit
            before pool start');
        require(block.timestamp < pool.endTime, 'Cannot deposit
            after pool ends');
```

```
        require(pool.totalDeposits < pool.maximumDeposit, 'Maximum deposit already reached');
        if (pool.totalDeposits.plus(amount) > pool.maximumDeposit) {
            amount = pool.maximumDeposit.minus(pool.totalDeposits);
        }
        pool.totalDeposits = pool.totalDeposits.plus(amount);
        pool.numReceipts = pool.numReceipts.plus(1);

        Receipt storage receipt = pool.receipts[pool.numReceipts];
        receipt.id = pool.numReceipts;
        receipt.amountDeposited = amount;
        receipt.timeDeposited = block.timestamp;
        receipt.owner = msg.sender;

        bool success = IERC20(pool.depositToken).transferFrom(msg.sender, address(this), amount);
        require(success, 'Token transfer failed');

        emit DepositOccurred(poolId, pool.numReceipts, msg.sender);
    }

```

3.6 (HAL-06) 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.

Code Location:

Listing 14: BasicPoolFactory.sol (Lines)

```
    function addPool (
        uint startTime,
        uint maxDeposit,
        uint[] memory rewardsPerSecondPerToken,
        uint programLengthDays,
        address depositTokenAddress,
        address[] memory rewardTokenAddresses
    ) public managementOnly {
        numPools = numPools.plus(1);
        Pool storage pool = pools[numPools];
        pool.id = numPools;
        pool.rewardsPerSecondPerToken = rewardsPerSecondPerToken;
        pool.startTime = startTime > block.timestamp ? startTime :
             block.timestamp;
        pool.endTime = startTime.plus(programLengthDays * 1 days);
        pool.depositToken = depositTokenAddress;
        require(rewardsPerSecondPerToken.length ==
            rewardTokenAddresses.length, 'Rewards and reward token
            arrays must be same length');

        for (uint i = 0; i < rewardTokenAddresses.length; i++) {
            pool.rewardTokens.push(rewardTokenAddresses[i]);
            pool.rewardsClaimed.push(0);
```

```
        }

        pool.maximumDeposit = maxDeposit;
    }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider declaring external variables instead of public variables. A best practice is to use external if expecting a function to only be called externally and public if called internally. Public functions are always accessible, but external functions are only available to external callers.

Remediation Plan:

SOLVED: Finance.Vote Team marked function as an external.

Listing 15: BasicPoolFactory.sol (Lines 60)

```
60        function addPool (
61        uint startTime,
62        uint maxDeposit,
63        uint[] memory rewardsPerSecondPerToken,
64        uint programLengthDays,
65        address depositTokenAddress,
66        address[] memory rewardTokenAddresses
67    ) external managementOnly {
68        numPools = numPools.plus(1);
69        Pool storage pool = pools[numPools];
70        pool.id = numPools;
71        pool.rewardsPerSecondPerToken = rewardsPerSecondPerToken;
72        pool.startTime = startTime > block.timestamp ? startTime :
              block.timestamp;
73        pool.endTime = startTime.plus(programLengthDays * 1 days);
```

```
        pool.depositToken = depositTokenAddress;
        require(rewardsPerSecondPerToken.length ==
            rewardTokenAddresses.length, 'Rewards and reward token
            arrays must be same length');

        for (uint i = 0; i < rewardTokenAddresses.length; i++) {
            pool.rewardTokens.push(rewardTokenAddresses[i]);
            pool.rewardsClaimed.push(0);
        }

        pool.maximumDeposit = maxDeposit;
        emit PoolAdded(pool.id);
    }
```

MANUAL TESTING

During the manual testing multiple questions where considered while evaluation each of the defined functions:

- Can it be re-called changing admin/roles and permissions?
- Can somehow an external controlled contract call again the function during the execution of it? (Re-entrancy)
- Do we control sensitive or vulnerable parameters?
- Does the function check for boundaries on the parameters and internal values? Bigger than zero or equal? Argument count, array sizes, integer truncation..
- Can an attacker withdraw multiple times?
- Can we deposit more than allowed?

4.1 Access Control Test

First of all, all contracts access control policies are evaluated. During the tests, the following functions are reachable by only management address.

Listing 16

```
function addPool (
        uint startTime,
        uint maxDeposit,
        uint[] memory rewardsPerSecondPerToken,
        uint programLengthDays,
        address depositTokenAddress,
        address[] memory rewardTokenAddresses
    ) public managementOnly
```

According to policies, No issues have been found on the dynamic analysis. Figure 1

MANUAL TESTING

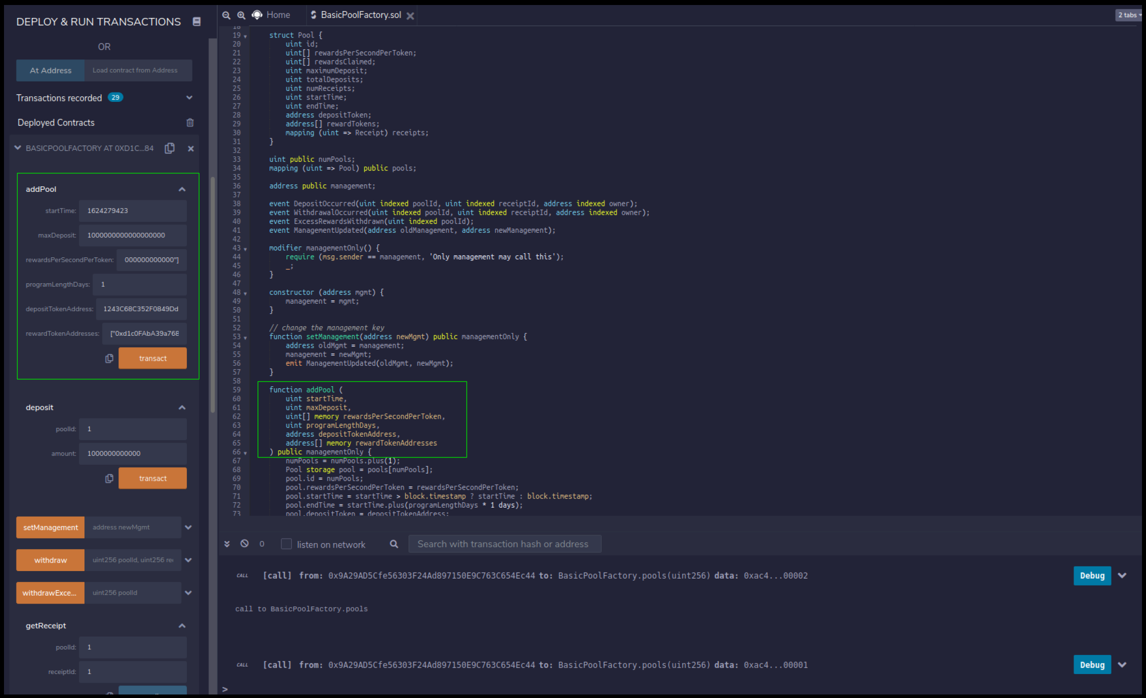


Figure 1: Testing Access Control Policy

4.2 Multiple Withdraw Test

Then, The withdraw progress has been tested. The Halborn Team tried to manipulate withdraw progress. Figure 2 From the test results, It has been observed that the user could not withdraw multiple times from the pools.

Next, Test cases ran on the contract functionalities. Multiple withdraw, owner checks are examined. Figure 3

- Screenshots

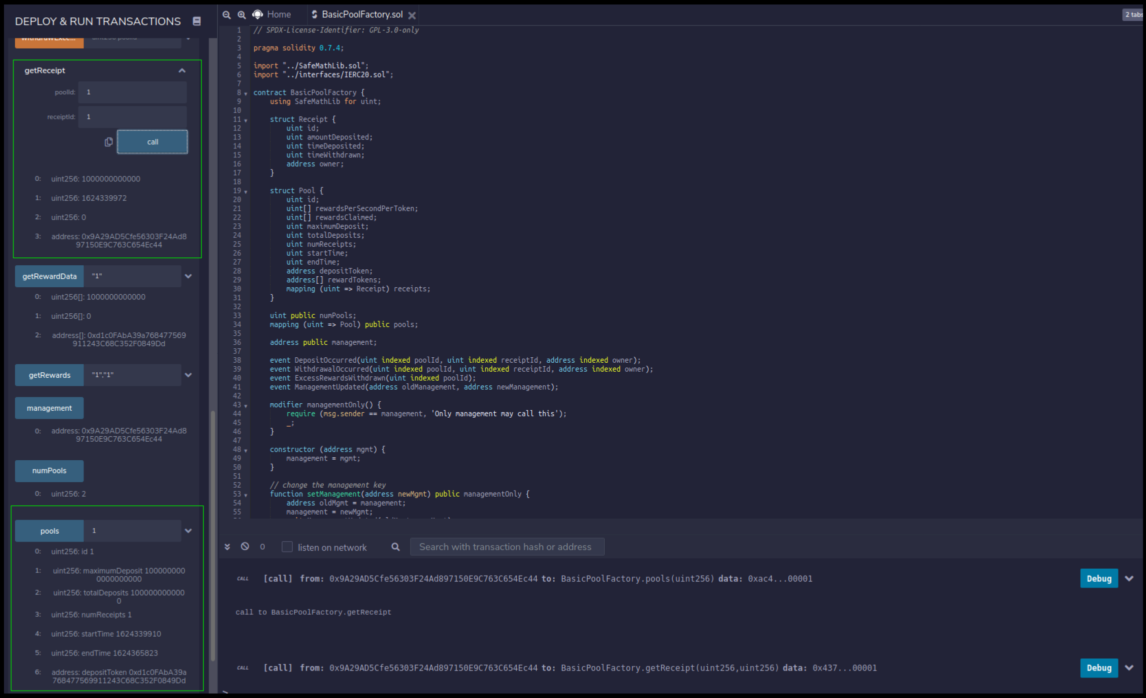


Figure 2: Example Pool Receipt

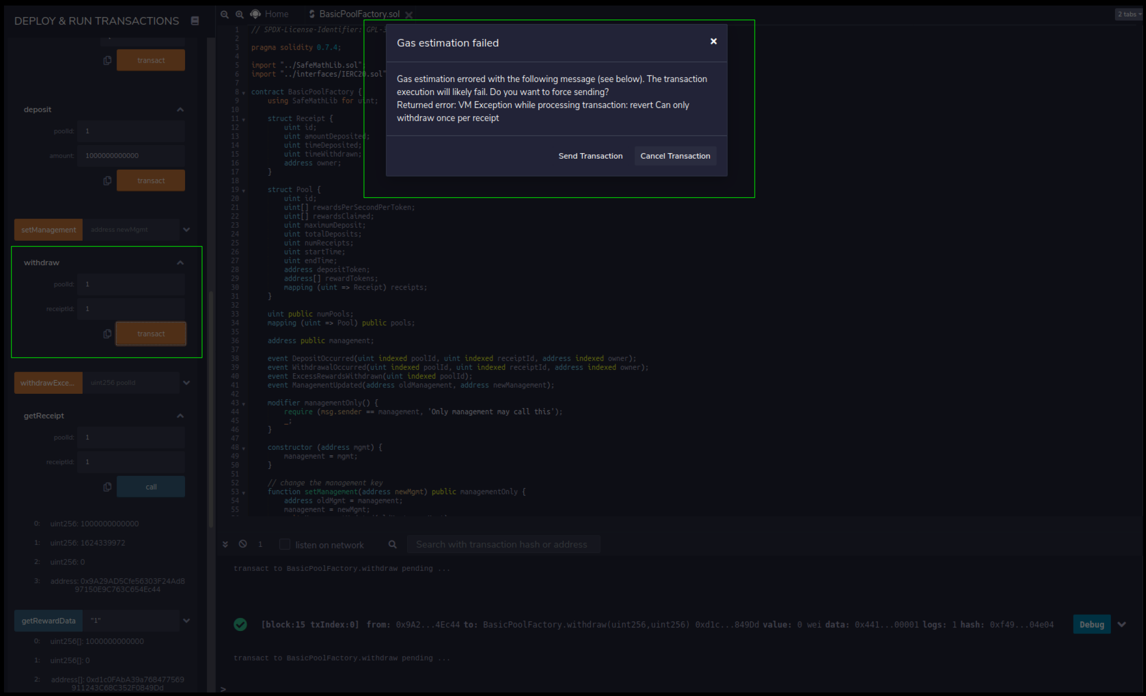


Figure 3: Multiple Withdraw Test

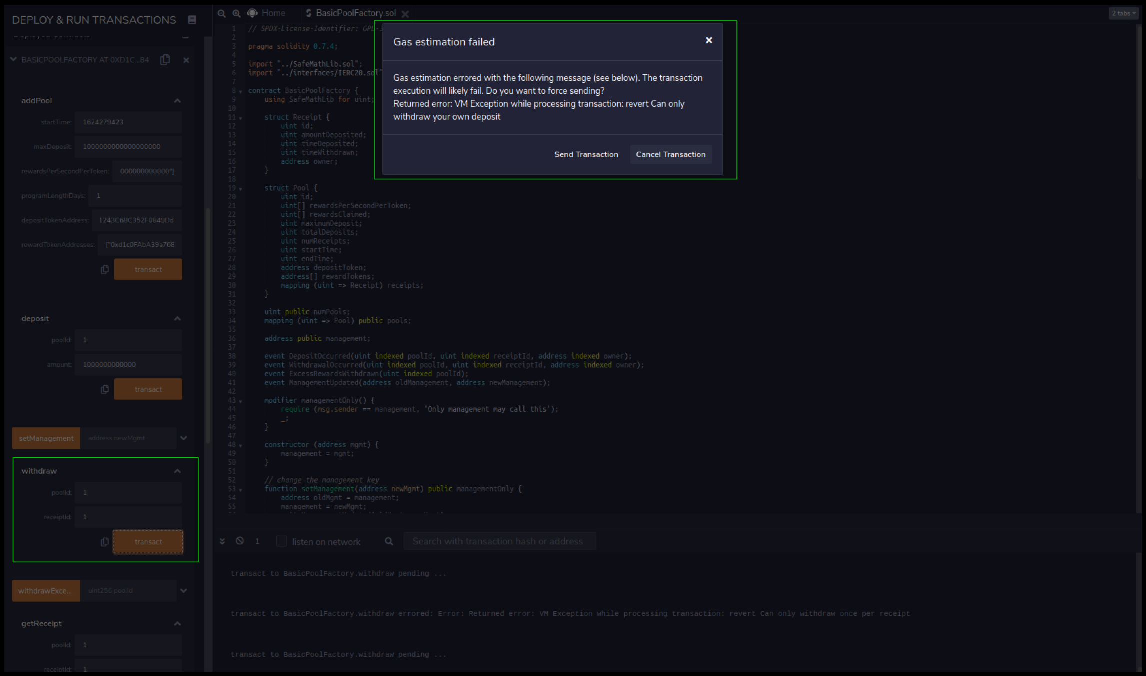


Figure 4: Receipt Owner Check

4.3 Reward Distribution Test

Rewards test

Test Code

Listing 17

```
function calculateRewards(uint startTime, uint endTime,uint rewardsPerSecondPerToken,uint amountDeposited) public view returns (uint) {
    uint secondsDiff = endTime.minus(startTime);
    uint rewardsLocal;
    rewardsLocal = (secondsDiff.times(rewardsPerSecondPerToken).times(amountDeposited)) / 1e18;

    return rewardsLocal;

}
```

Listing 18

```
Time Deposit Time - End Time   - Rewards Per Token - Amount
    Deposits (WEI) - Reward - Date Difference

1624452848 - 1624452849 - 10  - 1000000000000000 - 3 - 1 Second
1624452848 - 1627034048 - 10  - 1000000000000000 - 25812 - 1 Month
1624452848 - 1632390848 - 10  - 1000000000000000 - 79380 - 3 Month
1624452848 - 1640253248 - 10  - 1000000000000000 - 158004 - 6
    Month
1624452848 - 1655978048 - 10  - 1000000000000000 - 315252 - 1 Year
```

- Linear Graph

4.4 Deposit Amount Test

In that test case, Deposit amount is checked according to workflow. We tried to deposit to pool more than allowed. However, we are not successful for the manipulation.

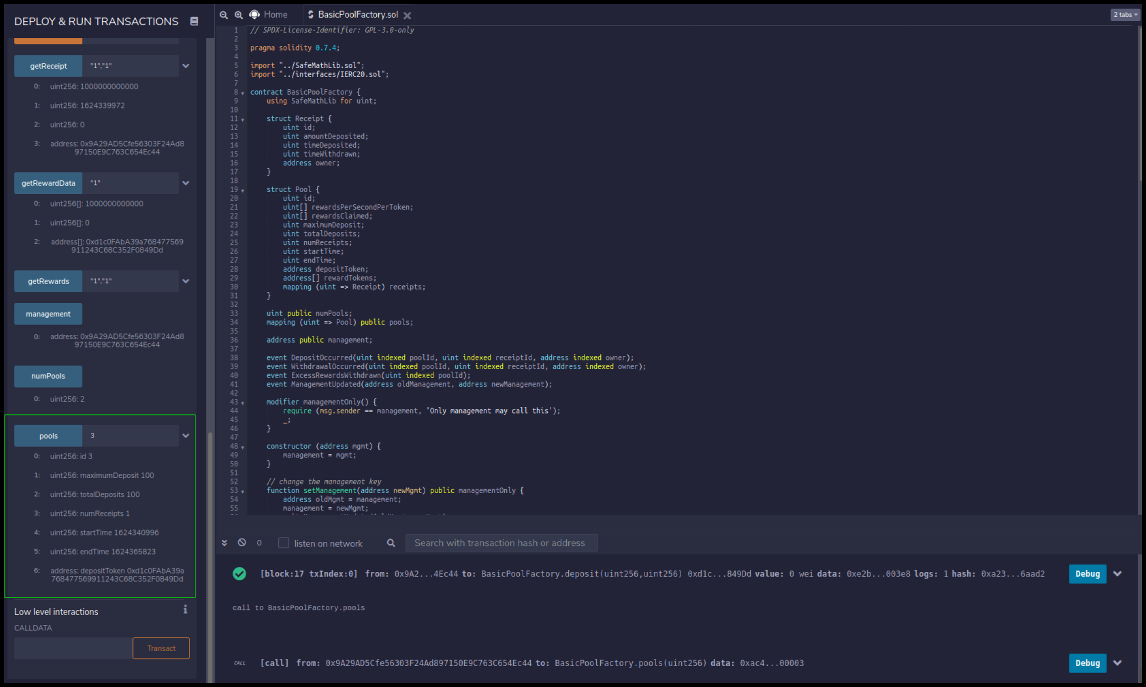


Figure 5: Deposit Amount Check

AUTOMATED TESTING

5.1 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, Slither was run on the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase.

Results:

BasicPoolFactory.sol

```
INFO:Detectors:
BasicPoolFactory (contracts/yield/BasicPoolFactory.sol#8-173) contract sets array length with a user-controlled value:
        - pool.rewardTokens.push(rewardTokenAddresses[i]) (contracts/yield/BasicPoolFactory.sol#77)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#array-length-assignment
INFO:Detectors:
Reentrancy in BasicPoolFactory.deposit(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#103-123):
        External calls:
        - IERC20(pool.depositToken).transferFrom(msg.sender,address(this),amount) (contracts/yield/BasicPoolFactory.sol#112)
        State variables written after the call(s):
        - pool.totalDeposits = pool.totalDeposits.plus(amount) (contracts/yield/BasicPoolFactory.sol#113)
        - pool.numReceipts = pool.numReceipts.plus(1) (contracts/yield/BasicPoolFactory.sol#114)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
BasicPoolFactory.deposit(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#103-123) ignores return value by IERC20(pool.depositToken).transferFrom(msg.sender,address(this),amount) (contr
acts/yield/BasicPoolFactory.sol#112)
BasicPoolFactory.withdraw(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#125-145) ignores return value by IERC20(pool.rewardTokens[i]).transfer(receipt.owner,rewards[i]) (contracts/yi
eld/BasicPoolFactory.sol#141)
BasicPoolFactory.withdraw(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#125-145) ignores return value by IERC20(pool.depositToken).transfer(receipt.owner,receipt.amountDeposited) (co
ntracts/yield/BasicPoolFactory.sol#143)
BasicPoolFactory.withdrawExcessRewards(uint256) (contracts/yield/BasicPoolFactory.sol#147-161) ignores return value by rewardToken.transfer(management,rewards) (contracts/yield/BasicPoolFacto
ry.sol#156)
BasicPoolFactory.withdrawExcessRewards(uint256) (contracts/yield/BasicPoolFactory.sol#147-161) ignores return value by depositToken.transfer(management,depositToken.balanceOf(address(this)))
(contracts/yield/BasicPoolFactory.sol#159)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
BasicPoolFactory.constructor(address).mgmt (contracts/yield/BasicPoolFactory.sol#48) lacks a zero-check on :
                - management = mgmt (contracts/yield/BasicPoolFactory.sol#49)
BasicPoolFactory.setManagement(address).newMgmt (contracts/yield/BasicPoolFactory.sol#53) lacks a zero-check on :
                - management = newMgmt (contracts/yield/BasicPoolFactory.sol#55)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
BasicPoolFactory.withdraw(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#125-145) has external calls inside a loop: IERC20(pool.rewardTokens[i]).transfer(receipt.owner,rewards[i]) (co
ntracts/yield/BasicPoolFactory.sol#141)
BasicPoolFactory.withdrawExcessRewards(uint256) (contracts/yield/BasicPoolFactory.sol#147-161) has external calls inside a loop: rewards = rewardToken.balanceOf(address(this)) (contracts/yiel
d/BasicPoolFactory.sol#155)
BasicPoolFactory.withdrawExcessRewards(uint256) (contracts/yield/BasicPoolFactory.sol#147-161) has external calls inside a loop: rewardToken.transfer(management,rewards) (contracts/yield/Basi
cPoolFactory.sol#156)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop
```

AUTOMATED TESTING

```
INFO:Detectors:
Reentrancy in BasicPoolFactory.deposit(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#103-123):
	External calls:
	- IERC20(pool.depositToken).transferFrom(msg.sender,address(this),amount) (contracts/yield/BasicPoolFactory.sol#112)
	Event emitted after the call(s):
	- DepositOccurred(poolId,pool.numReceipts,msg.sender) (contracts/yield/BasicPoolFactory.sol#122)
Reentrancy in BasicPoolFactory.withdraw(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#125-145):
	External calls:
	- IERC20(pool.depositToken).transfer(receipt.owner,receipt.amountDeposited) (contracts/yield/BasicPoolFactory.sol#143)
	Event emitted after the call(s):
	- WithdrawalOccurred(poolId,receiptId,receipt.owner) (contracts/yield/BasicPoolFactory.sol#144)
Reentrancy in BasicPoolFactory.withdrawExcessRewards(uint256) (contracts/yield/BasicPoolFactory.sol#147-161):
	External calls:
	- depositToken.transfer(management,depositToken.balanceOf(address(this))) (contracts/yield/BasicPoolFactory.sol#159)
	Event emitted after the call(s):
	- ExcessRewardsWithdrawn(poolId) (contracts/yield/BasicPoolFactory.sol#160)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
BasicPoolFactory.addPool(uint256,uint256,uint256[],uint256,address,address[]) (contracts/yield/BasicPoolFactory.sol#59-82) uses timestamp for comparisons
	Dangerous comparisons:
	- startTime > block.timestamp (contracts/yield/BasicPoolFactory.sol#71)
BasicPoolFactory.getRewards(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#84-101) uses timestamp for comparisons
	Dangerous comparisons:
	- nowish > pool.endTime (contracts/yield/BasicPoolFactory.sol#90)
BasicPoolFactory.deposit(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#103-123) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(block.timestamp > pool.startTime,Cannot deposit before pool start) (contracts/yield/BasicPoolFactory.sol#106)
	- require(bool,string)(block.timestamp < pool.endTime,Cannot deposit after pool ends) (contracts/yield/BasicPoolFactory.sol#107)
BasicPoolFactory.withdraw(uint256,uint256) (contracts/yield/BasicPoolFactory.sol#125-145) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(pool.id == poolId,Uninitialized pool) (contracts/yield/BasicPoolFactory.sol#127)
	- require(bool,string)(receipt.id == receiptId,Can only withdraw real receipts) (contracts/yield/BasicPoolFactory.sol#129)
	- require(bool,string)(receipt.owner == msg.sender || block.timestamp > pool.endTime,Can only withdraw your own deposit) (contracts/yield/BasicPoolFactory.sol#130)
	- require(bool,string)(receipt.timeWithdrawn == 0,Can only withdraw once per receipt) (contracts/yield/BasicPoolFactory.sol#131)
BasicPoolFactory.withdrawExcessRewards(uint256) (contracts/yield/BasicPoolFactory.sol#147-161) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(block.timestamp > pool.endTime,Contract must reach maturity) (contracts/yield/BasicPoolFactory.sol#151)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Pragma version0.7.4 (contracts/SafeMathLib.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version0.7.4 (contracts/interfaces/IERC20.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version0.7.4 (contracts/yield/BasicPoolFactory.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
solc-0.7.4 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
```

5.2 AUTOMATED SECURITY SCAN RESULTS

Description:

Halborn used automated security scanners to assist with detection of well known security issues, and identify low-hanging fruit on the scoped contract targeted 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 MythX to locate any vulnerabilities. Security Detections are only in scope, and the analysis was pointed towards issues with the BasicPoolFactory.sol.

Results:

Report for yield/BasicPoolFactory.sol
https://dashboard.mythx.io/#/console/analyses/a4ec2930-6e7b-43de-85b1-3e835a52e0f7

Line	SWC Title	Severity	Short Description
48	(SWC-100) Function Default Visibility	Low	Function visibility is not set.
53	(SWC-000) Unknown	Medium	Function could be marked as external.
59	(SWC-000) Unknown	Medium	Function could be marked as external.
70	(SWC-128) DoS With Block Gas Limit	Medium	Implicit loop over unbounded data structure.
96	(SWC-128) DoS With Block Gas Limit	Medium	Loop over unbounded data structure.
139	(SWC-128) DoS With Block Gas Limit	Medium	Loop over unbounded data structure.
153	(SWC-128) DoS With Block Gas Limit	Medium	Loop over unbounded data structure.
165	(SWC-128) DoS With Block Gas Limit	Low	Implicit loop over unbounded data structure.

Figure 6: Mythx results

All relevant findings were founded in the manual code review.

THANK YOU FOR CHOOSING

HALBORN