RED4SEC

Smart Contracts Security Audit

Babylon Finance

2021-06-14

Content

1. Introduction

Babylon Finance is a decentralized asset management protocol where funds are owned and led by the community. It is community owned and community managed. Trust-less and transparent.

Babylon Finance allows you to invest with the community that fits your risk, time, and liquidity preferences. Functions to participate directly with ETH with no complicated token swapping. Permits to retain ownership of your funds with non-custodial asset management.

As requested by Babylon Finance and as part of the vulnerability review and management process, Red4Sec has been asked to perform a security code audit and a cryptographic assessment in order to evaluate the security of the Babylon Finance Smart Contracts source code.

2. Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered neither "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project.

3. Scope

The Babylon Finance review includes the source code of the smart contracts in the following repository: github.com/babylon-finance/protocol, from commit `f4aaec570b2c1f06320b461a9d18aa12f8e063ab` and the remediations of these issues applied until commit `e01fcd70dbdca062eeaf09105d2545cb1b5e5ee2`.

4. Conclusions

The general conclusion of the audited contracts is that Babylon Finance's smart contracts **are secure and do not present any known vulnerabilities** that could compromise the security of the users. The overall impression about code quality and organization is very positive.

Some issues have been detected in the contracts that may affect their proper operation and they should be fixed before making the Smart Contracts deployment.

A few low impact issues were detected and classified only as informative, but they will continue to help Babylon Finance improving and optimizing quality of the project.

Babylon team has fixed the issues detailed in this report and their current status of the contract after the review done by the Red4Sec team is as follows:

Table of vulnerabilities

Id.	Vulnerability	Risk	State
BAB01	Wrong Owner Verification	Medium	Fixed
BAB02	Unbounded loop in _getLossesGarden methods	Medium	Fixed
BAB03	Wrong oracle price with same tokens	Informative	Dismissed
BAB04	Logic Mismatch	Low	Fixed
BAB05	Lack of inputs validation	Low	Partially Fixed
BAB06	Use of SafeMath to avoid integer overflows	Low	Fixed
BAB07	Arbitrary Modification of lastClaim	Informative	Fixed
BAB08	Underflow due to unexpected decimal tokens	Informative	Fixed
BAB09	Gas Optimization	Informative	Fixed
BAB10	Code Style	Informative	Fixed
BAB11	Outdated Compiler Version	Informative	Assumed

5. Issues & Recommendations

BAB01 - Wrong Owner Verification

The *getPrice* function contains code to verify and control only the invocations of authorized addresses. The issue is that the condition is written to always be accomplished, which makes the previous verifications useless and allows the invocation of the method to any address.

Even with a view-only method, the condition of the require does not accomplish the assertion. "Caller must be system contract".

```
function getPrice(address _assetOne, address _assetTwo) external view override returns (uint256) {
    require(
        // TODO: check is an strategy
        controller.isSystemContract(msg.sender) || msg.sender == owner() || true,
        'Caller must be system contract'
    );
```

This bypass is due to the development's tests of the team and being aware of this, the team has modified the method.

Source reference

- PriceOracle: 101

The remediation has been applied in the following pull request

- https://github.com/babylon-finance/protocol/pull/288

BAB02 - Unbounded Loop in _getLossesGarden methods

The logic executed to list all **finalized strategies** might trigger a denial of service (DoS) by GAS exhaustion because it iterates over the available **finalizedStrategies** without any limit.

Loops without limits are considered a bad practice in the development of Smart Contracts, since they can trigger a denial of service (DoS) or overly expensive executions, this is the case affecting Garden.

As there is no method to eliminate the completed strategies, once their profits have been distributed, there is a limitation in the number of completed strategies that force that the cost of iterating all the inputs surpasses the maximum allowed GAS per block, which currently is of 12 million approximately[1].

```
function _getLossesGarden(uint256 _since) private view returns (uint256) {
    uint256 totalLosses = 0;
    for (uint256 i = 0; i < finalizedStrategies.length; i++) {
        if (IStrategy(finalizedStrategies[i]).executedAt() >= _since) {
            totalLosses = totalLosses.add(IStrategy(finalizedStrategies[i]).getLossesStrategy());
        }
    }
    for (uint256 i = 0; i < strategies.length; i++) {
        if (IStrategy(strategies[i]).executedAt() >= _since) {
            totalLosses = totalLosses.add(IStrategy(strategies[i]).getLossesStrategy());
        }
    }

    return totalLosses;
}
```

Source references

- Garden:1051

The remediation has been applied in the following pull request

- https://github.com/babylon-finance/protocol/pull/270

BAB03 - Wrong oracle price with same tokens

The *getPrice* method of the **PriceOracle** contract produces a disparity if two equal tokens are entered with a different number of decimals than expected (18). This method assumes that when it is the same token, the decimals are 18 in both cases and it should be noted that not all tokens have the same number of decimals, such as USDC which has 6 decimals.

```
function getPrice(address _assetOne, address _assetTwo) external view override returns (uint256) {
    require(controller.isSystemContract(msg.sender) || msg.sender == owner(), 'Caller must be system contract');
    // Same asset. Returns base unit
    if (_assetOne == _assetTwo) {
        return 10**18;
    }
```

In the previous image we can observe how if we compare USDC vs USDC we would obtain an amount that results incorrect.

[1] https://ethgasstation.info/blog/ethereum-block-size

This vulnerability has been categorized as medium, although if we analyze its impact in isolation it would be higher, but considering the Babylon environment the exploitability is drastically reduced due to the governance of the project.

Source references

- PriceOracle:114

Resolution

After further research with the Babylon team, it has been concluded that it is the expected behavior to be able to operate with the Uniswap pools, which normalizes the decimals to 18. Since the contract is used for this purpose, the risk of this issue can be ruled out.

BAB04 - Logic Mismatch

The logic of the *claimReturns* method in the **Garden** contract does not contemplate all the possible cases, so in case there are profits and some balance, but not enough to distribute the profits, the method will fail.

```
if (totalProfits > 0 && address(this).balance > 0) {
    contributor.claimedProfits = contributor.claimedProfits.add(totalProfits);
    // Send ETH
    Address.sendValue(msg.sender, totalProfits);
    profitsSetAside = profitsSetAside.sub(totalProfits);
    emit ProfitsForContributor(msg.sender, totalProfits);
    contributor.claimedAt = block.timestamp; // Checkpoint of this claim
}
```

Source references

- Garden:468

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/ee26d1641f12003c035388ae1ffbc37edb1b120a

The *_withdraw* method checks if the contract has enough Ether as net flow, and if this is not the case, it converts wETH to Ether. The problem is that when Ether is withdrawn, it does not consider the current balance of the contract and it converts more than the necessary amount.

```
// Check that the withdrawal is possible
// Unwrap WETH if ETH balance lower than netFlowQuantity
if (address(this).balance < withdrawalInfo.netFlowQuantity) {
    IWETH(WETH).withdraw(withdrawalInfo.netFlowQuantity);
}
```

Source references

- Garden:1020

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/ee26d1641f12003c035388ae1ffbc37edb1b120a

BAB05 - Lack of inputs validation

Some methods of the different contracts in the Babylon project do not properly check the arguments, which can lead to major errors. Below we list the most significant examples.

The *register* method of the **TimeLockRegistry** does not check that the vesting start date *(vestingStartingDate)* is not a past date, allowing to add vesting with past dates.

Source references

- TimeLockRegistry: 118

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/7ddc156ba80d267dded49d03d8cc0cf07f40dd98

RewardsDistributor of *getSupplyForPeriod* does not correctly verify the periods and it allows the *_from* to be less than a *_to.*

Source references

- RewardsDistributor: 382

The remediation has been applied in the following changes

- https://github.com/babylon-finance/protocol/commit/e019c7d44102946eeb5463eded789609ac56d14f
- https://github.com/babylon-finance/protocol/pull/293

The *setGardenAccess* method properly verifies that the address is not *address(0)*, however the *setGardenAccessBatch* method does not contain the same verification.

```
function setGardenAccess(
    address _user,
    address _garden,
    uint8 _permission
) external override onlyGardenCreator(_garden) returns (uint256) {
    require(address(_user) != address(0), 'User must exist');
    return _setIndividualGardenAccess(_user, _garden, _permission);
}

/**...
function grantGardenAccessBatch(
    address _garden,
    address[] calldata _users,
    uint8[] calldata _perms
) external override onlyGardenCreator(_garden) returns (bool) {
    require(_users.length == _perms.length, 'Permissions and users must match');
    for (uint8 i = 0; i < _users.length; i++) {
        _setIndividualGardenAccess(_users[i], _garden, _perms[i]);
    }
    return true;
}
```

Source references

- IshtarGate: 125

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/ee26d1641f12003c035388ae1ffbc37edb1b120a

The *setData* method of the **LiquidityPoolStrategy** contract should verify that *poolTokens* is greater than zero, which would produce unwanted errors in methods such as *_enterStrategy.*

Source references

- LiquidityPoolStrategy: 52

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/d45fca3e338b938e8df19b463af321358366c7d8

The **Garden** contract does not check in its initializer that the size of *_gardenPrams* is the size expected 9.

Source references

- Garden:280

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/ee26d1641f12003c035388ae1ffbc37edb1b120a

BAB06 - Use of SafeMath in order to avoid Integer overflows

The Open Zeppelin SafeMath class is properly used throughout the contract to protect the contract from incorrect or malicious arithmetic operations. However, in the *getEpochRewards* function on line 359 it performs a direct subtraction with the arithmetic operator instead of using SafeMath's subtraction safe method.

An example of this issue can be found in: RewardDistributor.sol:359 It should be noted that, although it is a good practice, the current implementation is safe and has a lower consumption of GAS, as long as the staked token works as expected.

Source references

- RewardDistributor:359

The remediation has been applied in the following pull request

- https://github.com/babylon-finance/protocol/pull/293

```
function getEpochRewards(uint256 epochs) external pure override returns (uint96[] memory) {
    uint96[] memory tokensPerEpoch = new uint96[](epochs);
    for (uint256 i = 0; i <= epochs - 1; i++) {
        tokensPerEpoch[i] = (uint96(tokenSupplyPerQuarter(i.add(1))));
    }
    return tokensPerEpoch;
}
```

In the observationIndexOf function, *epochPeriod* performs a division without the use of safeMath.

Source references

- UniswapTAP:113

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/818f7230cefe0a87969423237dd24610219cc417

The *computeAmountOut* function does not use safeMath in the operations, therefore it is vulnerable to overflows.

```
function computeAmountOut(
    uint256 priceCumulativeStart,
    uint256 priceCumulativeEnd,
    uint256 timeElapsed
) private pure returns (uint256 amountOut) {
    // overflow is desired.
    FixedPoint.uq112x112 memory priceAverage =
        FixedPoint.uq112x112(uint224((priceCumulativeEnd - priceCumulativeStart) / timeElapsed));
    amountOut = priceAverage.mul(1).decode144();
    amountOut = amountOut.preciseDiv(1);
}
```

Source references

- UniswapTAP:179

In the *getLockedBalance* function of the **Garden** contract, SafeMath is not used in arithmetic operations so it is vulnerable to overflows.

```
function getLockedBalance(address _contributor) external view override returns (uint256) {
    uint256 lockedAmount;
    for (uint256 i = 0; i <= strategies.length - 1; i++) {
        IStrategy strategy = IStrategy(strategies[i]);
        uint256 votes = uint256(Math.abs(strategy.getUserVotes(_contributor)));
        if (votes > 0) {
            lockedAmount += votes;
        }
        if (_contributor == strategy.strategist()) {
            lockedAmount += strategy.stake();
        }
    }
    if (balanceOf(_contributor) < lockedAmount) lockedAmount = balanceOf(_contributor); //
    return lockedAmount;
}
```

Source references

- Garden:865

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/620aa262c851d2d34477ff369461914cb36a45a4#diff-a3c290b6e20efc6a732ac98c0c945adb9e7fcfa2492bc05c932f23674756919c

Recommendations

- It is advisable to establish the types in an explicit manner, to avoid the assumption that the types are identical and to use an index that could have been overflowed when performing a conversion.
- Choose an integer type used for a variable that is consistent with the functions to be performed or that can hold all the possible values of an arithmetic operation.
- Both operands and results of an integer operation should be validated and checked for overflow conditions.
- Additionally, use known and audited libraries such as SafeMath.sol from OpenZeppelin to avoid overflows and mathematical operations issues.

BAB07 - Arbitrary Modification of lastClaim

The *lockedBalance* method of the **TimeLockedToken** contract allows for any user to modify the *lastClaim* of any *vestedToken* registry to the current date. The method does not check that the address that invokes the contract is the same as the one which is modifying it.

```
function lockedBalance(address account) public returns (uint256) {
    // get amount from distributions locked tokens (if any)

    uint256 lockedAmount = viewLockedBalance(account);

    // in case of vesting has passed, all tokens are now available so we set mapping to 0
    if (block.timestamp >= vestedToken[account].vestingEnd && msg.sender == account && lockedAmount == 0) {
        delete distribution[account];
    } else {
        vestedToken[account].lastClaim = block.timestamp;
    }
    return lockedAmount;
}
```

VestedToken.lastClaim is a simple registry with no impact on the rest of the contract's logic, therefore, it is a good idea to remove this uncontrolled assignment from *account* when the condition is false.

Source references

- TimeLockedToken:274

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/2c775236c102e39c4e4fc77517c369f64f4e8781

BAB08 - Underflow due to unexpected decimal tokens

The Open Zeppelin SafeMath contract is properly used throughout the code to protect the contract from incorrect or malicious arithmetic operations. However, in the *_normalizeDecimals* function of the **Operation** contract it performs a direct subtraction with the arithmetic operator instead of using SafeMath's subtraction safe method, this operation produces an underflow with tokens that have decimals greater than 18.

Currently there are no assets allowed in the whitelist that have more than 18 decimals, however it is possible that in the future it will be allowed to add new assets and become affected.

```
function _normalizeDecimals(address _asset, uint256 _quantity) internal view returns (uint256) {
    uint8 tokenDecimals = ERC20(_asset).decimals();
    return tokenDecimals != 18 ? _quantity.mul(10**(18 - tokenDecimals)) : _quantity;
}
```

Source references

- Operation:130

The remediation has been applied in the following pull request

- https://github.com/babylon-finance/protocol/pull/350/files

BAB09 - Gas Optimization

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On Ethereum blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.

- Delete unnecessary libraries.
- Explicit function visibility.
- Use of proper data types.
- Use hard-coded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to mathematical operations and comparisons.

Logic Optimizations

Unlike the previous cases, this optimization affects all the variables and not just during the deployment. So, by optimizing this function the cost of GAS in each transaction will be lower, saving the users costs in GAS.

The value of upper (nCheckpoints - 1) of the VoteToken can be previously calculated and reused whenever said value is needed, instead of calculating it every time.

```
// First check most recent balance
if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
    return checkpoints[account][nCheckpoints - 1].votes;
}

// Next check implicit zero balance
if (checkpoints[account][0].fromBlock > blockNumber) {
    return 0;
}

uint32 lower = 0;
uint32 upper = nCheckpoints - 1;
```

Source reference

- VoteToken:169

In the **PriceOracle** contract, the variables *symbol1* and *symbol2* of the *_getPriceFromUniswapAnchoredView* method should only be declared if the logic of the conditional that uses them is accomplished, right before its used, so it is advisable to move said declaration in order to perform an optimization of GAS in certain cases.

```
string memory symbol1 = _assetOne == WETH ? 'ETH' : ERC20(_assetOne).symbol();
string memory symbol2 = _assetTwo == WETH ? 'ETH' : ERC20(_assetTwo).symbol();
address assetToCheck = _assetOne;
if (_assetOne == WETH) {
    assetToCheck = _assetTwo;
}
if (
    assetToCheck == 0x6B175474E89094C44Da98b954EedeAC495271d0F || // dai
    assetToCheck == 0x1985365e9f78359a9B6AD760e32412f4a445E862 || // rep
    assetToCheck == 0xE41d2489571d322189246DaFA5ebDe1F4699F498 || // zrx
    assetToCheck == 0x0D8775F648430679A709E98d2b0Cb6250d2887EF || // bat
    assetToCheck == 0xdd974D5C2e2928deA5F71b9825b8b646686BD200 || // knc
    assetToCheck == 0x514910771AF9Ca656af840dff83E8264EcF986CA || // link
    assetToCheck == 0xc00e94Cb662C3520282E6f5717214004A7f26888 || // comp
    assetToCheck == 0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984 // uni
) {
    uint256 assetOnePrice = IUniswapAnchoredView(uniswapAnchoredView).price(symbol1);
    uint256 assetTwoPrice = IUniswapAnchoredView(uniswapAnchoredView).price(symbol2);

    if (assetOnePrice > 0 && assetTwoPrice > 0) {
        return (true, assetOnePrice.preciseDiv(assetTwoPrice));
    }
}
```

Source reference

- PriceOracle:182

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/ee26d1641f12003c035388ae1ffbc37edb1b120a

It has been detected that in the *canWithdrawEthAmount* function of the **Garden** contract the declaration of the *ethAsideBalance* variable is made before its use, it should be declared only if the logic of the conditional that uses it is fulfilled, right before its use.

```
function canWithdrawEthAmount(address _contributor, uint256 _amount) public view returns (bool) {
    uint256 ethAsideBalance = address(this).balance;
    uint256 liquidWeth = IERC20Upgradeable(reserveAsset).balanceOf(address(this));

    // Weth already available
    if (liquidWeth >= _amount) {
        return true;
    }

    // Withdrawal open
    if (block.timestamp <= withdrawalsOpenUntil) {
        // Pro rata withdrawals
        uint256 contributorPower =
            _getContributorPower(_contributor, contributors[_contributor].initialDepositAt, block.timestamp);
        return ethAsideBalance.preciseMul(contributorPower) >= _amount;
    }
    return false;
}
```

Source reference

- Garden:764

Redundant Code

The assignment of the variable *vestedToken[_receiver]* is unnecessary as it has already been set on line 169.

```
VestedToken storage newVestedToken = vestedToken[_receiver];

newVestedToken.teamOrAdvisor = _profile;
newVestedToken.vestingBegin = _vestingBegin;
newVestedToken.vestingEnd = _vestingEnd;
newVestedToken.lastClaim = _lastClaim;

vestedToken[_receiver] = newVestedToken;
```

Source reference

- TimeLockedToken:176

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/d4db10c86d1c6db0330f4369a10721e8589fed09

The *sendTreasuryFunds* method of the **Treasury** contract performs a double verification of the balance of the contract, first in the require of *balanceOf* and again in the own logic of the *safeTransfer*.

```
function sendTreasuryFunds(
    address _asset,
    uint256 _amount,
    address _to
) external onlyOwner {
    require(_asset != address(0), 'Asset must exist');
    require(_to != address(0), 'Target address must exist');
    require(IERC20(_asset).balanceOf(address(this)) >= _amount, 'Not enough funds in treasury');
    IERC20(_asset).safeTransferFrom(address(this), _to, _amount);
    emit TreasuryFundsSent(_asset, _amount, _to);
}
```

Source reference

- Treasury:76

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/ee26d1641f12003c035388ae1ffbc37edb1b120a

In the *removeAdapter* function of **PriceOracle** contract, it is verified that the address of the *adapter* is included in the array.

```
function removeAdapter(address _adapter) external onlyOwner {
    require(adapters.contains(_adapter), 'Adapter does not exist');
    adapters = adapters.remove(_adapter);

    emit AdapterRemoved(_adapter);
}
```

However, as we can check in the following image, this comparison is not necessary since the *remove* function verifies internally, this redundant code can result in an additional GAS expense.

```
/**
 * @param A The input array to search
 * @param a The address to remove
 * @return Returns the array with the object removed.
 */
function remove(address[] memory A, address a) internal pure returns (address[] memory) {
    (uint256 index, bool isIn) = indexOf(A, a);
    if (!isIn) {
        revert('Address not in array.');
    } else {
        (address[] memory _A, ) = pop(A, index);
        return _A;
    }
}
```

Source references

- PriceOracle:139

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/ee26d1641f12003c035388ae1ffbc37edb1b120a

Variable Optimization

The use of constants is recommended as long as the variables are never to be modified. In this case the variables "*vestingCliff*", "*teamVesting*" and "*investorVesting*" of the **TimeLockedToken** contract should be declared as constants since they would not be necessary to access the storage to read the content of these variables and therefore the execution cost is much lower.

Also, those variables are not currently being used, so if they become necessary it would be convenient to convert them to constants, as it should be in the case of the **TimeLockRegistry** contract.

Source references

- TimeLockedToken: 97-103
- TimeLockRegistry: 84-87

The remediation has been applied in the following pull request

- https://github.com/babylon-finance/protocol/pull/357

Another example of an unused variable is the *_from* variable in the **Garden** contract's *_getContributorPower* function.

Source reference

- Garden:1190

The remediation has been applied in the following pull request

- https://github.com/babylon-finance/protocol/pull/291

In some cases, a few repeated operations have been found, where saving the result would avoid repeating these operations with the consequent gas saving. It becomes more important in operations where external methods are invoked that imply an extra cost of gas. Below are some cases present in *GovernorAlpha* and *RewardsDistributor* Contract

In **GovernorAlpha** the loop in line 271 *proposal.targets.length* can be cached to avoid calculating it for each iteration of the loop.

At **RewardsDistributor** we also find repeated operations like those listed below.

1. *substractProtocolPrincipal*

```
// Any other strategy different from the very first one (will have an antecesor)
protocolCheckpoint.power = protocolPerTimestamp[timeList[pid.sub(1)]].power.add(
    protocolCheckpoint.time.sub(protocolPerTimestamp[timeList[pid.sub(1)]].time).mul(
        protocolPerTimestamp[timeList[pid.sub(1)]].principal
    )
);
```

2. *getStrategyRewards*

```
uint256[] memory strategyPower = new uint256[](numQuarters);
uint256[] memory protocolPower = new uint256[](numQuarters);
for (uint256 i = 0; i <= numQuarters.sub(1); i++) {
```

3. *_addProtocolPerQuarter*

```
function _addProtocolPerQuarter(uint256 _time) private {
    ProtocolPerQuarter storage protocolCheckpoint = protocolPerQuarter[getQuarter(_time)];

    if (!isProtocolPerQuarter[getQuarter(_time).sub(1)]) {
        // The quarter is not yet initialized then we create it
        protocolCheckpoint.quarterNumber = getQuarter(_time);
```

4. *_updatePowerOverhead*

```
function _updatePowerOverhead(IStrategy _strategy, uint256 _capital) private {
    if (_strategy.updatedAt() != 0) {
        // There will be overhead after the first execution not before
        if (getQuarter(block.timestamp) == getQuarter(_strategy.updatedAt())) {
            // The overhead will remain within the same epoch
            rewardsPowerOverhead[address(_strategy)][getQuarter(block.timestamp)] = rewardsPowerOverhead
                address(_strategy)
            ][getQuarter(block.timestamp)]
                .add(_capital.mul(block.timestamp.sub(_strategy.updatedAt())));
```

Source reference

- RewardsDistributor: 174, 202, 244, 716, 808

The remediation has been applied in the following pull request

- https://github.com/babylon-finance/protocol/pull/351

Storage Optimizations

The use of the *immutable* keyword is recommended to obtain less expensive executions, by having the same behaviour as a constant. However, by defining its value in the constructor we have a significant save of GAS.

This behaviour has been found across several contracts, so it should be reviewed to find possible optimizations.

Also, it should be mentioned that the access to storage in Solidity is an extremely expensive process, so it should always be optimized as much as possible. Through caching variables, we can avoid querying the storage; or through 'storage' keyword we can obtain a pointer, both actions will help to have lower cost transactions.

In this way, in the **TimeLockRegistry** smart contract, by caching the result of access to the *tokenVested[]* storage on the *claim* method, a considerable gas saving can be obtained.

```
// get amount from distributions
uint256 amount = registeredDistributions[_receiver];
tokenVested[_receiver].lastClaim = block.timestamp;

// set distribution mapping to 0
delete registeredDistributions[_receiver];

// register lockup in TimeLockedToken

// this will transfer funds from this contract and lc
token.registerLockup(
    _receiver,
    amount,
    tokenVested[_receiver].team,
    tokenVested[_receiver].vestingBegin,
    tokenVested[_receiver].vestingEnd,
    tokenVested[_receiver].lastClaim
);
```

Source references

- TimeLockRegistry: 240, 268
- RewardsDistributor: 426, 476
- Garden: 1215-1217

Deleting unnecessary registers once they are used is less expensive than keeping them at their default value. This is the case of *validReserveAsset* in the *removeReserveAsset* method.

```
function removeReserveAsset(address _reserveAsset) external override onlyOwner {
    require(validReserveAsset[_reserveAsset], 'Reserve asset does not exist');

    reserveAssets = reserveAssets.remove(_reserveAsset);

    validReserveAsset[_reserveAsset] = false;

    emit ReserveAssetRemoved(_reserveAsset);
}
```

Source references

- BabController:297

Memory accesses are always more optimal than storage accesses, so whenever it's possible, we must prioritize the use of memory over the access to the variables of the storage.

```
function setDelay(uint256 delay_) external {
    require(msg.sender == address(this), 'Timelock::setDela
    require(delay_ >= MINIMUM_DELAY, 'Timelock::setDelay: D
    require(delay_ <= MAXIMUM_DELAY, 'Timelock::setDelay: D
    delay = delay_;

    emit NewDelay(delay);
}

function acceptAdmin() external override {
    require(msg.sender == pendingAdmin, 'Timelock::acceptAd
    admin = msg.sender;
    pendingAdmin = address(0);

    emit NewAdmin(admin);
}

function setPendingAdmin(address pendingAdmin_) external {
    require(msg.sender == address(this), 'Timelock::setPend
    pendingAdmin = pendingAdmin_;

    emit NewPendingAdmin(pendingAdmin);
}
```

Source references

- TimeLock: 96, 104,111

BAB10 - Code Style

These are not vulnerabilities by itself but improving them will help to improve the code and reduce the appearance of new vulnerabilities.

As a reference, it is always recommendable to apply some coding style/good practices that can be found in multiple standards such as:

- "Solidity Style Guide" (https://docs.soliditylang.org/en/v0.8.0/style-guide.html).

These references are very useful to improve smart contract quality. Some of those practices are common and a popular accepted way to develop software.

In the **BabController** contract, specifically in the *disableGarden* function a double negation has been detected, that might not return the expected value to the user.

```
function disableGarden(address _garden) external override onlyOwner {
    require(isGarden[_garden], 'Garden does not exist');
    IGarden garden = IGarden(_garden);
    require(!!garden.active(), 'The garden needs to be active.');
    garden.setDisabled();
}
```

Source references

- BabController:218

The remediation has been applied in the following commit

- https://github.com/babylon-finance/protocol/commit/ee26d1641f12003c035388ae1ffbc37edb1b120a

BAB11 - Outdated Compiler Version

Solc frequently launches new versions of the compiler. Using an outdated version of the compiler can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect this version.
We have detected that the audited contract uses the following version of Solidity pragma *0.7.4*:

```
pragma solidity 0.7.4;
```

It is always of good policy to use the most up to date version of the pragma.

Solidity branch *0.7* has important bug fixes in the array processing, so it is recommended to use the most up to date version of the pragma.

References

- https://github.com/ethereum/solidity/blob/develop/Changelog.md

RED4SEC

Invest in Security, invest in your future