byterocket

Orbit DeFi

May 29, 2022

1. Preface

The developers of **Orbit DeFi** contracted byterocket to conduct a smart contract audit of their token contract suite. Orbit is "*a DeFi smart vault Optimizer that automates and rebalances your LP strategies effortlessly, starting with Uniswap V3*".

The team of byterocket reviewed and audited the above smart contracts in the course of this audit. We started on the 6th of May and finished on the 29th of May 2022.

The audit included the following services:

- *Manual Multi-Pass Code Review*
- *Protocol/Logic Analysis*
- *Automated Code Review*
- *Formal Report*

byterocket gained access to the code via a public GitHub repository. We based the audit on the main branch's state on May 12th, 2022 (*commit hash 541460999b7d8fa80c4a4a86383f583392436b67*). The updated version was provided to us via multiple new commits to the repository, addressing our findings. The last and most recent commit hash that we audited is *3c6ed8c356be28b2e059a2a7f0171ba31f216bb7*.

2. Manual Code Review

We conducted a manual multi-pass code review of the smart contracts mentioned in section (1). Three different people went through the smart contract independently and compared their results in multiple concluding discussions.

The manual review and analysis were additionally supported by multiple automated reviewing tools, like Slither, GasGauge, Manticore, and different fuzzing tools.

2.1 Severity Categories

We are categorizing our findings into four different levels of severity:

Non-Critical	Does not impose immediate risk but is relevant to security best practices. Includes issues with - Code style and clarity - Versioning - Off-chain monitoring
Low Severity	Imposes relatively small risks or could impose risks in the long-term but without assets being at risk in the current implementation. Includes issues with - State handling - Functions being incorrect as to specification - Faulty documentation or in-code comments
Medium Severity	Imposes risks on the function or availability of the protocol or imposes financial risk by leaking value from the protocol if external requirements are met.
High Severity	Imposes catastrophic risk for users and/or the protocol. Includes issues that could result in - Assets being stolen/lost/compromised - Contracts being rendered useless - Contracts being gained control of

2.2 Summary

On the code level, we **found 39 bugs or flaws, with 39 of them being fixed** in a subsequent update**.** Prior to this, there have been 19 non-critical and 6 low, 9 medium and 5 high severity findings. Additionally, we found **9 gas improvements**, which have **all been implemented**.

The contracts are written according to the latest standard used within the Ethereum community and the Solidity community's best practices. The naming of variables is very logical and understandable, which results in the contract being easy to understand. The code is well documented. The developers provided us with a test suite as well as proper deployment scripts.

2.3 Findings

H.1 - Prevent contract to be initialized multiple times [HIGH SEVERITY] [FIXED]

Location: PositionManager.sol - Line 105 - 116

Description:
The PositionManager can be re-inititialized countless times, as the init() function as well as the construction function does not implement any way to check whether the contract has been initialized already. This way, ownership can easily be claimed via the init() function.

Recommendation:
Consider adding a flag to ensure that the contracts can only be initialized once. Additionally, we suggest making use of the OpenZeppelin libraries for initialization processes.

Update on the 13th of June 2022:
The developers have update the implementation to make use of OpenZeppelin's initializer, preventing multiple initializations.

H.2 - SwapToPositionRatio function can be called by anyone [HIGH SEVERITY] [FIXED]

Location: actions/SwapToPositionRatio.sol - Line 34 - 75

Description:
The swapToPositionRatio() function has no checks on authorization, hence it can be called and executed by anyone at any time. There might be cases, where this is not in the best interest of the user, especially during sudden price swings out of

the current range.

The same applies to the following functions:

- modules/AaveModule.sol - depositIfNeeded() in line 37
- modules/AaveModule.sol - withdrawIfNeeded() in line 55
- modules/AutoCompoundModule.sol - autoCompoundFees() in line 30
- modules/IdleLiquidityModule.sol - rebalance() in line 31 (Can not be called by anyone, but might still not be good!)

Recommendation:
Consider adding a proper check for the correct authorization of the swapToPosition- Ratio() function.

Update on the 14th of June 2022:
The developers have updated their implementation to include the onlyWhitelisted- Keepers modifier, except for the swapToPositionRatio() function, where it is not necessary to do so. This action function is - as the developers told us - only being called via a fallback function that is properly protected. Subsequently, we consider this finding fixed.

H.3 - LendingPoolAddress can be set by anyone [HIGH SEVERITY] [FIXED]

Location: utils/AaveAddressHolder.sol - Line 16 - 18

Description:
The setLendingPoolAddress() function has no limitations on who can call and execute it and thus change the lending pools address.

These are the affected lines of code:

```
function setLendingPoolAddress(address newAddress) external override {
 lendingPoolAddress = newAddress;
}
```

Recommendation:
Consider adding a proper check to ensure that only the right addresses can change the lending pool address.

Update on the 13th of June 2022:
The developers have update the implementation to only allow governance to change these variables.

H.4 - LendingPoolAddress can be set by anyone [HIGH SEVERITY] [FIXED]

Location: actions/AaveDeposit.sol - Line 28 - 30

Description:
As the lending pool address of the Aave action can be set by anyone (due to (H.3)), users may deposit into a malicious lending pool or the lending pool of an attacker without knowing so.

These are the affected lines of code:

```
PositionManagerStorage.getStorage().aave[...].lendingPoolAddress()
```

Recommendation:
Consider addressing (H.3) or finding another way to ensure that the original lending address that a user defined can not be changed without them noticing or them intending to do so.

Update on the 13th of June 2022:
As the developers have properly address finding (H.3), this finding has subsequently been addressed as well.

H.5 - UniswapAddressHolder variables can be set by anyone [HIGH SEVERITY] [FIXED]

Location: utils/UniswapAddressHolder.sol - Lines 25, 31, 37

Description:
The variables (nonfungiblePositionManagerAddress, uniswapV3FactoryAddress, and swapRouterAddress) have no limitations on who can set them - thus anyone can do it at any time.

This is one of the affected functions:

```
function setNonFungibleAddress(address newAddress) external override {
 nonfungiblePositionManagerAddress = newAddress;
}
```

Recommendation:
Consider adding a proper check to ensure that only the right addresses can change the variables.

Update on the 13th of June 2022:
The developers have update the implementation to only allow governance to change these variables.

M.1 - Access protection is not working in modifier [MEDIUM SEVERITY] [FIXED]

Location: PositionManager.sol - Line 90

Description:
The onlyFactory(_registry) modifier is not working as intended, as its input can be controlled by the caller. The calling address can deploy its own registry with registry.positionManagerFactoryAddress == msg.sender.

Furthermore, the user might be able to deploy their own address this way, but still make use of the official one.

These are the affected lines of code:

```
constructor(
  address _owner,
  address _diamondCutFacet,
  address _registry
) payable onlyFactory(_registry) {
  [...]
}
```

Recommendation:
Consider changing the access control to the PositionManager to be safe and not have any user-controlled inputs that can change its authorization.

Update on the 13th of June 2022:
The developers have revamped the constructor-flow as well as the initialization process. The new flow does not contain any user-controller inputs as far as authorization goes, which now leads to the initialization to work as expected.

M.2 - False approval mechanism in AaveDeposit [MEDIUM SEVERITY] [FIXED]

Location: actions/AaveDeposit.sol - Line 41 - 43

Description:
The deposit function of the Aave action contains a false approval/allowance mechanism. Firstly, it is important to first set the allowance to zero before changing it to a custom value, since some tokens enforce it (see here). Secondly, we would suggest only approving the correct amount (diff) instead of the full amount.

This subsequently also occurs in helpers/ERC20Helper.sol in lines 22 to 25.

These are the affected lines of code:

```
if (IERC20(token).allowance(address(this), address(lendingPool)) <  amount) {
  IERC20(token).approve(address(lendingPool), amount);
}
```

Recommendation:
Consider changing the approval mechanism to first set the allowance to zero before changing it, as well as only approving the diff instead of the full amount.

Update on the 13th of June 2022:
The developers have update the implementation to first set the allowance to zero before updating it to the required amount.

M.3 - Swap is not safe for front-runs [MEDIUM SEVERITY] [FIXED]

Location: actions/Swap.sol - Line 50

Description:
During the swap, the call defines the input as well as the expected output. Setting the expected output to zero opens the door to a huge front-running opportunity as the arbitrageur has maximum room for price changes for this trade. Setting the proper expected output with a sensible slippage applied is within the best practice.

This also occurs in the following places:

- actions/SwapToPositionRatio.sol - Line 101
- actions/ZapOut.sol - Line 97 (*with 1 Wei instead of 0*)

These are the affected lines of code:

```
ISwapRouter.ExactInputSingleParams memory swapParams = ISwapRouter.ExactInputSingleParams({
  tokenIn: token0Address,
  tokenOut: token1Address,
  fee: fee,
  recipient: address(this),
  deadline: block.timestamp + 120,
  amountIn: amount0In,
  amountOutMinimum: 0,
  sqrtPriceLimitX96: 0
});
```

Recommendation:
Consider making use of a proper amount for the expected out tokens instead of setting it to 0.

Update on the 14th of June 2022:
The developers have implemented a new time-weighted average pricing deviation check in the SwapHelper file, which they are using to ensure that the swaps are not carried out when they would incur a loss that is bigger than the accepted deviation. Subsequently, we consider this finding fixed.

M.4 - Logic to find best pool does not find best one [MEDIUM SEVERITY] [FIXED]

Location: actions/ZapOut.sol - Line 111 - 120

Description:
The corresponding section in the _findBestFee() function intended to find the best pool is actually just returning the pool with the most liquidity at the given slot. However, if the impact of the swap leads to the liquidity of the current slot being used up, the rest of the swap is being facilitated in a slot whose liquidity is unknown to the function.

This also occurs in modules/AaveModule.sol in lines 214 to 222.

These are the affected lines of code:

```
for (uint8 i = 0; i < 4; i++) {
 try this.getPoolLiquidity(token0, token1, uint24(fees[i])) returns
 (uint128 nextLiquidity) {
   if (nextLiquidity > bestLiquidity) {
     bestLiquidity = nextLiquidity;
     fee = fees[i];
   }
 } catch {
   //pass
 }
}
```

Recommendation:
Consider either documenting this behaviour or simulating the trade in each of the pools to really find out, which of the offers the best circumstances for the corresponding trade.

Update on the 13th of June 2022:
The developers have update the documentation to reflect the described behaviour.

M.5 - Missing liquidity in Aave deposit [MEDIUM SEVERITY] [FIXED]

Location: modules/AaveModule.sol - Line 103 - 128

Description:
The amountToAave that is being calculated in the corresponding code section only accounts for the collected fees, not for the liquidity received by “closing” the position.

Recommendation:
Consider including the liquidity that has been received as well, instead of just handling the fees.

Update on the 14th of June 2022:
The developers have responded to our finding, stating that the liquidity is included. It is taken into account by being stored in the tokensOwed variable after a call to decreaseLiquidity(). Afterwards, the liquidity is considered for the collectFees() call. Subsequently, we consider this finding fixed.

M.6 - Relying on external security promises for ticks [MEDIUM SEVERITY] [FIXED]

Location: modules/IdleLiquidityModule.sol - Line 114 - 118

Description:
The result of the call to the NonfungiblePositionManager returns the lower and upper ticks. These are not checked and can lead to the calculation in line 111 to over- or underflow, without erroring as no SafeMath is used. They could be in the wrong order as well if not properly checked.

These are the affected lines of code:

```
(, , , , , int24 tickLower, int24 tickUpper, , , , , ) =
INonfungiblePositionManager(uniswapAddressHolder.nonfungiblePositionManagerAddress()).positions(tokenId);
int24 tickDelta = tickUpper - tickLower;
```

Recommendation:
Consider verifying whether the ticks are actually ordered correctly as well as making use of SafeMath in this case as well to ensure that no under- or overflow can occur unnoticed.

Update on the 13th of June 2022:

The developers have update the implementation to make use of SafeMath, ensuring that no unnoticed under- or overflows can occur.

M.7 - Unsafe cast to int24 [MEDIUM SEVERITY] [FIXED]

Location: modules/IdleLiquidityModule.sol - Line 129

Description:
The cast of a uint24 to int24 is dangerous and should be properly executed and verified, which is not the case here. This can lead to severely wrong values that are being subsequently used.

These are the affected lines of code:

```
int24 tickSpacing = int24(fee) / 50;
```

Recommendation:
Consider properly verifying whether the outcome of the unsafe cast is correct.

Update on the 13th of June 2022:
The developers have update the implementation to include a library, which handles the safe casting between int24 and uint24. This library is now being used.

M.8 - Severe over-/underflow risk without SafeMath [MEDIUM SEVERITY] [FIXED]

Location: modules/IdleLiquidityModule.sol - Line 131

Description:
As SafeMath is not used, there are several risks (see (L.3)). In this case, there is a severe risk of under- and/or overflows due to the nature of the calculation. Additionally, (in the latter part of the calculation) dividing by tickSpacing and then multiplying by tickSpacing only reduces accuracy without doing anything meaningful, which might not be the desired behavior.

The same occurs in utils/WithdrawRecipes.sol in lines 49 - 50 and in modules/AutoCompoundModule.sol in lines 72 - 73.

These are the affected lines of code:

```
return (((tick - tickDelta) / tickSpacing) * tickSpacing, ((tick + tickDelta) / tickSpacing) * tickSpacing);
```

Recommendation:
Consider making use of SafeMath and removing unnecessary parts of the calculation.

Update on the 13th of June 2022:
The developers have update the implementation to make use of SafeMath.

M.9 - Empty data doesn’t always revert [MEDIUM SEVERITY] [FIXED]

Location: modules/IdleLiquidityModule.sol - Line 40

Description:
With the modules and actions generally reverting in case of the data being non-existent, this is mostly fine. However, in this case, a definitive rebalance is being facilitated as the values here might still pass the if-checks.

Recommendation:
Consider properly checking whether the module’s data has been correctly set.

Update on the 13th of June 2022:
The developers have update the implementation to ensure that the data of the module is not set to zero.

L.1 - Wrong comparison operator [LOW SEVERITY] [FIXED]

Location: helpers/ERC20Helper.sol - Line 54

Description:
During the check on whether the balance is greater than the required amount, a *less than* (<) is used, instead of making use of a *less than equal* (<=).

These are the affected lines of code:

```
if (amount - balance < _getBalance(token, from)) {
 needed = amount - balance;
 IERC20(token).safeTransferFrom(from, address(this), needed);
}
```

Recommendation:
Consider changing the comparison to reflect a *less than equal* (<=) instead of a *less than* (<) to correct it.

Update on the 13th of June 2022:

The developers have update the implementation to use a *less than equal* (<=) instead of a *less than* (<).

L.2 - Misleading or wrong documentation [LOW SEVERITY] [FIXED]

Location: Multiple occurrences throughout the project

Description:

Certain parts of the documentation are misleading or wrong:

- helpers/ERC20Helper.sol - Line 62 - 65
- -> The function does not withdraw the specified amount if the user does not have that many tokens
- modules/IdleLiquidityModule.sol - Line 39
- -> tickDistance can be smaller than zero
- utils/Storage.sol - Line 190
- -> The function does not return but instead reverts in this case

Recommendation:
Consider correcting the documentation to ensure that they are correct and not misleading.

Update on the 13th of June 2022:
The developers have update the documentation to better represent the actual logic of the code.

L.3 - Not making use of SafeMath where appropriate [LOW SEVERITY] [FIXED]

Location: Multiple occurrences throughout the project

Description:
In certain locations, calculations are facilitated without SafeMath in a Solidity version that is lower than 0.8:

- helpers/SwapHelper.sol -> Throughout the whole contract
- actions/ZapIn.sol - Lines 53 - 54

Recommendation:
Consider using SafeMath functionalities in the corresponding occurrences to ensure their required safety guarantees.

Update on the 13th of June 2022:
The developers have update the implementation to make use of SafeMath in the corresponding occurrences.

L.4 - Faulty reliance on data to be present [LOW SEVERITY] [FIXED]

Location: Multiple occurrences throughout the project

Description:
As modules currently do not require any data to be set, it is not the best idea to rely on the data being present. In the cases that we found, the call would still revert, but we do not recommend this handling of missing data. This occurs in the following cases:

- modules/AaveModule.sol - depositIfNeeded() in line 37
- modules/AutoCompoundModule.sol - _checkIfCompoundIsNeeded() in line 50
- modules/IdleLiquidityModule.sol - rebalance() in line 31

Recommendation:
Consider adding a proper handling to functions that interact with a module while relying on its data to be present. This can also just be a proper require statement.

Update on the 13th of June 2022:
The developers have update the implementation to ensure that the data of the module is not zero.

L.5 - Use of 32 bits does not save gas [LOW SEVERITY] [FIXED]

Location: utils/DepositRecipes.sol - Line 36

Description:
The use of a uint32 variable does not save gas in this case, as you can see here.

The same also applies to the use of uint8 for the i-variable in for-loops, like in modules/AaveModule.sol in line 213.

These are the affected lines of code:

```
for (uint32 i = 0; i < tokenIds.length; i++) {
 [...]
}
```

Recommendation:
Consider reverting the counter back to uint256 to ensure that it can not overflow.

Update on the 13th of June 2022:
The developers have update the implementation to use uint256 variables instead of uint32.

L.6 - Use of unsafe ERC20 transfers [LOW SEVERITY] [FIXED]

Location: Multiple occurrences throughout the project

Description:
Throughout the project, the unsafe versions of ERC20 transfers are used. With the variety of todays tokens, especially malicious ones, this is never recommended. This occurs in the following cases:

- utils/DepositRecipes.sol - Lines 69 - 70
- actions/AaveDeposit.sol - Line 42 (*approve*)
- helpers/ERC20Helper.sol - Line 25 (*approve*)

Recommendation:
Consider making use of SafeERC20 mechanics to ensure a safe transfer of ERC20 tokens.

Update on the 13th of June 2022:
The developers have update the implementation to make use of SafeERC20 in all of the corresponding occurrences.

NC.1 - Non-standard delete from array pattern [NO SEVERITY] [FIXED]

Location: PositionManager.sol - Line 128 - 137

Description:
A non-standard way of "delete from array" pattern is being used here, which we do not usually recommend.

These are the affected lines of code:

```
for (uint256 i = 0; i < uniswapNFTs.length; i++) {
 if (uniswapNFTs[i] == tokenId) {
   if (uniswapNFTs.length > 1) {
     uniswapNFTs[i] = uniswapNFTs[uniswapNFTs.length - 1];
     uniswapNFTs.pop();
   } else {
     delete uniswapNFTs;
   }
   return;
 }
}
```

Recommendation:
Consider changing the pattern to be more standardized to be safe, like in the example below:

```
for (uint256 i = 0; i < uniswapNFTs.length; i++) {
 if (uniswapNFTs[i] == tokenId) {
   if (i + 1 != uniswapNFTs.length) {
     uniswapNFTs[i] = uniswapNFTs[uniswapNFTs.length - 1];
   }
   uniswapNFTs.pop();
   return;
 }
}
```

Update on the 13th of June 2022:
The developers have update the implementation to make use of more a standardized pattern to remove an element from the array.

NC.2 - Key should be private to enforce proper use [NO SEVERITY] [FIXED]

Location: utils/Storage.sol - Line 38

Description:
To ensure a proper usage of the getStorage() function, the key variable should be set to private, so it can not be accessed on accident.

These are the affected lines of code:

```
bytes32 constant key = keccak256('position-manager-storage-location');
```

Recommendation:
Consider adding the private keyword to the key variable to ensure a proper use of the getStorage() function.

Update on the 13th of June 2022:
The developers have update the variable to be private, enforcing the proper use of their function.

NC.3 - Undocumented constant value [NO SEVERITY] [FIXED]

Location: modules/IdleLiquidityModule.sol - Line 70

Description:
There is an undocumented value of 10 being deducted from the swapped token amounts. Furthermore, this value is not being defined as a global (constant) variable for better readability.

These are the affected lines of code:

```
IMint.MintInput(token0, token1, fee, tickLower, tickUpper, token0Swapped - 10, token1Swapped - 10)
```

Recommendation:
Consider documenting the value of 10 and possibly converting it to a global (constant) variable.

Update on the 13th of June 2022:
The developers have update the variable itself as well as its documentation to make sure that it can be properly understood.

NC.4 - Invalid documentation [NO SEVERITY] [FIXED]

Location: PositionManager.sol - Line 257 - 258

Description:
The documentation of the withdrawERC20() function does state that it returns tokens, while in reality it transfers them to the msg.sender.

Recommendation:
Consider correcting the documentation to reflect the actual function.

Update on the 13th of June 2022:
The developers have update the documentation to reflect the actual logic of the function.

NC.5 - Wrong use of allowance [NO SEVERITY] [FIXED]

Location: PositionManager.sol - Line 260

Description:
The withdrawERC20() function includes an approveToken call, which is (a) not required at all, (b) approving the wrong address, as well as (c) using the wrong amount for the approval.

Additionally, using type(uint256).max is best practice instead of using 2**256 - 1.

These are the affected lines of code:

```
ERC20Helper._approveToken(tokenAddress, address(this), 2**256 - 1);
uint256 amount = ERC20Helper._withdrawTokens(tokenAddress, msg.sender,
            2**256 - 1);
```

Recommendation:
Consider removing the approve-call and change the uint256 maximum value to reflect the best practice of Solidity. We would suggest a change like the following:

```
uint amount = ERC20Helper._getBalance(tokenAddress, address(this));
uint got = ERC20Helper._withdrawTokens(tokenAddress, msg.sender, amount);
require(amount == got, "ERC20 Transfer failed");
emit ERC20Withdrawn(tokenAddress, msg.sender, got);
```

Update on the 13th of June 2022:
The developers have update the implementation, removing the unnecessary approval statement and making use of our suggested change.

NC.6 - Use underscores for large numerals [NO SEVERITY] [FIXED]

Location: utils/WithdrawRecipes.sol - Line 31, 49, 50

Description:
It is best practice to write numerals that are 1,000 or higher with underscores, like so:

- 1_000 instead of 1000
- 17_521_395 instead of 17521395

This highly increases its readability.

Recommendation:
Consider adding underscores to numerals to that are 1,000 or higher.

Update on the 13th of June 2022:
The developers have update the implementation, adding underscored to the corresponding numerals.

NC.7 - Not using the getStorage function [NO SEVERITY] [FIXED]

Location: PositionManager.sol - Line 294

Description:
The fallback() function retrieves the storage manually instead of making use of the provided getStorage() function.

These are the affected lines of code:

```
StorageStruct storage Storage;
bytes32 position = PositionManagerStorage.key;
///@dev get diamond storage position
assembly {
 Storage.slot := position
}
```

Recommendation:
Consider making use of the getStorage() function instead of doing it manually.

Update on the 13th of June 2022:
The developers have update the implementation to enforce the proper use of the getStorage() function.

NC.8 - Unnecessary double function exposed [NO SEVERITY] ACKNOWLEDGED

Location: PositionManagerFactory.sol - Line 86 - 88

Description:
The getAllPositionManagers() function merely returns the positionManagers array, which is unnecessary as the array is public anyways. This is increasing the contract size without any benefit.

The same also applies to the moduleKeys array in the Registry, with the getModuleKeys() function in line 102 being implemented as well.

Recommendation:
Consider either removing the getAllPositionManagers() function or changing the array to be private to prevent two exposed functions to exist.

Update on the 14th of June 2022:
The developers have stated that they have a good reason to keep the extra function, which is perfectly fine for us. There is no security implication be keeping it.

NC.9 - Unnecessary deadline extension [NO SEVERITY] [FIXED]

Location: PositionManagerFactory.sol - Line 46 - 49

Description:
The changeGovernance() function has two slight problems that we see: it does not validate its input, which - in this case - might be crucial. Additionally, it might emit the event in cases where there has been no change to the state.

The same also applies to the same function in the Registry.sol contract in line 47 - 50.

Recommendation:
Consider validating the input of crucial functions like this one to be valid. Additionally, ensure that events for state-changing functions are only emitted if the state variable has actually been changed.

Update on the 13th of June 2022:
The developers have update the implementation to validate whether the supplied address is not zero.

NC.10 - Insufficient state-modifying function [NO SEVERITY] [FIXED]

Location: actions/ClosePosition.sol - Line 46

Description:
The deadline of the swap is being extended by 120 seconds, even though it is being executed by a contract instead of an EOA, in which case it is unnecessary.

The same occurs in the following places:

- actions/CollectFees.sol - Line 59

- actions/DecreaseLiquidity.sol - Line 76
- actions/IncreaseLiquidity.sol - Line 58
- actions/Mint.sol - Line 57
- actions/SwapToPositionRatio.sol - Line 99
- actions/ZapOut.sol - Line 43
- actions/ZapOut.sol - Line 95

These are the affected lines of code:

```
deadline: block.timestamp + 120
```

Recommendation:
Consider removing the 120 second addition, as it is unnecessary in this case.

Update on the 13th of June 2022:
The developers have update the implementation, removing the unnecessary 120 second addition to the deadline.

NC.11 - Wrong type used for Uniswap V3 [NO SEVERITY] [FIXED]

Location: actions/ClosePosition.sol - Line 50 - 52

Description:
The token0Closed and token1Closed variables are of type uint128, as defined in the Uniswap V3 contracts. However, in your implementation, you are using the uint256 type instead, which leads to an implicit conversion.

These are the affected lines of code:

```
(, , , , , , , , , , uint256 token0Closed, uint256 token1Closed) = nonfungiblePositionManager.positions(tokenId);
```

Recommendation:
Consider changing the type of the two variables to be uin128 and then (if required) convert it to uint256 to ensure that no side-effect occur. During our internal tests, this still worked as intended, but we can't guarantee it in any case.

Update on the 13th of June 2022:
The developers have update the implementation to work with uint128 for the return values of the Uniswap V3 calls.

NC.12 - Not using the default method for maximum values [NO SEVERITY] [FIXED]

Location: actions/ClosePosition.sol - Line 57 - 58

Description:
Solidity has a default way to ensure a clean and error-free way of setting the maximum value of a type with type(uint128).max instead of doing a manual calculation. This is not being used here.

The same occurs in the following places:

- actions/Swap.sol - Lines 40 - 41
- actions/SwapToPositionRatio - Lines 91 - 92
- actions/ZapOut.sol - Lines 51 - 52

These are the affected lines of code:

```
INonfungiblePositionManager.CollectParams memory collectparams = INonfungiblePositionManager.CollectParams({
 tokenId: tokenId,
 recipient: returnTokenToUser ? Storage.owner : address(this),
 amount0Max: 2**128 - 1,
 amount1Max: 2**128 - 1
});
```

Recommendation:
Consider making use of type(uint128).max instead of 2**128-1.

Update on the 13th of June 2022:
The developers have update the implementation throughout the project to make use of the type(uint).max flow instead of manually calculations.

NC.13 - Making use of undocumented side-effect [NO SEVERITY] ACKNOWLEDGED

Location: actions/CollectFees.sol - Line 51 - 64

Description:
The _updateUncollectedFees() function does look like a rather costly way of updating the uncollected fee through an undocumented side-effect. This is usually not within the best practice, but does certainaly not cause any harm here.

Recommendation:
We leave it up to the developers on whether they want to keep using this behaviour. We just wanted to point it out in case this behaviour changes over time.

Update on the 14th of June 2022:
The developers have acknowledged the finding and are open to find a better solution. As there is currently no better solution that is known to either of us, it is perfectly fine to stay like this.

NC.14 - Only emit events if the state changed [NO SEVERITY] [FIXED]

Location: actions/SwapToPositionRatio.sol - Line 74

Description:
It is best practice to only emit an event if the state has actually been changed. This usually does require an additional if-statement to be added, but is generally accepted to be more consistent in terms of the on-chain history.

This also occurs in the following locations:

- utils/Storage.sol - Line 62

Recommendation:
Consider adding an if-statement to ensure that the event is only emitted, if something was actually changed.

Update on the 13th of June 2022:
The developers have update the implementation to only emit events in cases where something was actually changed.

NC.15 - Relying on false guarantees [NO SEVERITY] [FIXED]

Location: actions/ZapIn.sol - Line 45

Description:
The _pullTokensIfNeeded() function does not guarantee that any tokens have actually been pulled. In this case, it does not look like anything would happen, but this is not within the best practice.

Recommendation:
Consider calls to functions like _pullTokensIfNeeded() when writing your logic and keep in mind that they are not required to actually do what they say. We would suggest that an additional require statement may be used to ensure that everything has worked out.

Update on the 13th of June 2022:
The developers have update the implementation to ensure that any call to _pullTokensIfNeeded() only succeeds if it really did its job properly.

NC.16 - Use of transferFrom instead of transfer [NO SEVERITY] [FIXED]

Location: helpers/ERC20Helper.sol - Line 77

Description:
The _withdrawTokens() function transfers tokens from itselfv to an address via transferFrom, which is not correct. This can be handled with a regular transfer.

These are the affected lines of code:

IERC20(token).safeTransferFrom(address(this), to, amountOut);

Recommendation:
Consider changing the transferFrom to a regular transfer.

Update on the 13th of June 2022:
The developers have update the implementation to make use of a regular transfer instead of a transferFrom.

NC.17 - Unchangeable global variables should be immutable [NO SEVERITY] [FIXED]

Location: Throughout the project

Description:
There are certain occurrences of global variables that should never change, hence should be immutable to ensure that this is the case as well as saving some gas. These occurrences are:

- PositionManager.sol - Line 12 - 15
- modules/AaveModule.sol - Lines 21 - 22
- modules/AutoCompoundModule.sol - Line 17
- modules/IdleLiquidityModule.sol - Line 19

- utils/DepositRecipes.sol - Line 17 - 18
- utils/WithdrawRecipes.sol - Line 18 - 19
- modules/BaseModule..sol - Line 10

Recommendation:
Consider adding the immutable keyword to the corresponding global variables that qualify.

Update on the 13th of June 2022:
The developers have update the implementation, setting the corresponding unchangeable global variables to be immutable.

NC.18 - Missing visibility keyword [NO SEVERITY] [FIXED]

Location: Throughout the project

Description:
There are certain occurrences of global variables that are missing a visibility keyword, which leads to them being private, which is often not the desired behaviour. These occurrences are:

- PositionManager.sol - Line 13 - 14
- modules/AaveModule.sol - Line 22
- modules/AutoCompoundModule.sol - Line 17
- utils/DepositRecipes.sol - Line 17 - 18
- utils/WithdrawRecipes.sol - Line 18 - 19

Recommendation:
Consider adding the correct visibility keyword to the corresponding global variables that are missing one.

Update on the 13th of June 2022:
The developers have update the implementation, adding a visibility keyword to the corresponding global variables.

NC.19 - Function can be exported to library [NO SEVERITY] [FIXED]

Location: modules/AaveModule.sol - Line 173

Description:
The code of the distance check is a near duplicate of the code in the _getTokens() function of the UniswapNFTHelper contract. This code can be exported into a library

Recommendation:
Consider introducing a library to prevent unnecessary code duplication.

Update on the 13th of June 2022:
The developers have moved the logic into the UniswapNFTHelper contract.

2.4 Gas Optimizations

DONE - GO.1 - Cache array length if accessed inside of for-loop

Location: Throughout the project

Description:
There are several for-loops that can benefit from caching the length of the used array, which is often the case if it is being accessed outside of the declaration of the loop itself. Caching the array length outside a loop saves reading it on each iteration, as long as the array's length is not changed during the loop.

The affected for-loops are:

- PositionManager.sol - Line 128
- PositionManager.sol - Line 271
- PositionManager.sol - Line 284
- utils/DepositRecipes.sol - Line 36
- utils/Storage.sol - Line 81
- utils/Storage.sol - Line 106
- utils/Storage.sol - Line 177
- utils/Storage.sol - Line 192

This is an example of an affected loop:

```
for (uint256 i = 0; i < uniswapNFTs.length; i++) {
 if (uniswapNFTs[i] == tokenId) {
   if (uniswapNFTs.length > 1) {
     uniswapNFTs[i] = uniswapNFTs[uniswapNFTs.length - 1];
     uniswapNFTs.pop();
   } else {
     delete uniswapNFTs;
   }
   return;
 }
}
```

Recommendation:
Consider moving the length of the used array into a cached variable like in the example below:

```
uint256 nfts = uniswapNFTs.length;
for (uint256 i = 0; i < nfts; i++) {
 if (uniswapNFTs[i] == tokenId) {
   if (uniswapNFTs.length > 1) {
     uniswapNFTs[i] = uniswapNFTs[nfts - 1];
 [...]
}
```

Update on the 14th of June 2022:
The developers have updated their implementation, taking our suggestion for gas improvements into consideration.

DONE - GO.2 - Optimize loops to be more efficient

Location: Throughout the project

Description:
All of the for-loops in the project are using the standard way of declaring them, which is slightly inefficient gas-wise.

This is an example of an unoptimized for-loop:

```
for (uint256 i = 0; i < array.length; i++) {
 [...]
}
```

Recommendation:
Consider optimizing the for-loops in each contract to be more efficient in terms of its gas usage, including:

- Not initializing variables with their default value
- Using the more efficient way to increment a variable

```
for (uint256 i; i < array.length; ++i) {
 [...]
}
```

Update on the 14th of June 2022:
The developers have updated their implementation, taking our suggestion for gas improvements into consideration.

DONE - GO.3 - Don’t initialize variables with default value

Location: Throughout the project

Description:
Initializing variables with their default values uses more gas than necessary as it involves an unnecessary MSTORE or SSTORE operation.

The affected variables are:

- actions/ZapOut.sol - Line 108
- helpers/ERC20Helper.sol - Line 51
- modules/AaveModule.sol - Line 113
- modules/AaveModule.sol - Line 222

This is an example of an affected variable:

```
uint128 bestLiquidity = 0;
```

Recommendation:
Consider removing occasions where the default value is being used to initialize a variable to save some gas, as shown in the example below:

```
uint128 bestLiquidity;
```

Update on the 14th of June 2022:
The developers have updated their implementation, taking our suggestion for gas improvements into consideration.

DONE - GO.4 - Use !=0 instead of >0 for uint comparisons

Location: Throughout the project

Description:
In **require** statements, it is cheaper to check for != 0 instead of > 0, if the unsigned integer is being validated to not be zero. Changing this will save some gas.

The affected variables are:

- PositionManager.sol - Line 237
- helpers/SwapHelper.sol - Line 46
- utils/Storage.sol - Line 98
- utils/Storage.sol - Line 169
- utils/Storage.sol - Line 188
- utils/Storage.sol - Line 203
- utils/WithdrawRecipes.sol - Line 34

This is an example of an affected require statement check:

```
require(amount0In > 0 || amount1In > 0);
```

Recommendation:
Consider changing all of the require statements that involved a zero check for a unsigned integer to be more gas efficient, like in the example below:

```
require(amount0In != 0 || amount1In != 0);
```

Update on the 14th of June 2022:
The developers have updated their implementation, taking our suggestion for gas improvements into consideration.

DONE - GO.5 - Cache certain variables to save gas

Location: Throughout the project

Description:
In certain circumstances, it saves gas to cache variables that are read often, especially if they are stored in storage and not in memory.

The affected variables are:

- PositionManager.sol - Line 237 (*aaveUserReserves*)
- actions/AaveDeposit.sol - Line 33 (*aTokenAddress*)
- actions/DecreaseLiquidity.sol - Line 44 (*nonfungiblePositionManagerAddress*)
- actions/IncreaseLiquidity.sol - Line 37 (*nonfungiblePositionManagerAddress*)
- actions/Mint.sol - Line 37 (*nonfungiblePositionManagerAddress*)
- actions/ZapIn.sol - Line 123 (*nonfungiblePositionManagerAddress*)
- actions/ZapOut.sol - Line 86 (*swapRouterAddress*)
- modules/AaveModule.sol - Line 71 (*nonfungiblePositionManagerAddress*)
- modules/AaveModule.sol - Line 95 (*nonfungiblePositionManagerAddress*)
- modules/AutoCompoundModule.sol - Line 59 (*nonfungiblePositionManagerAddress*)
- modules/IdleLiquidityModule.sol - Line 126 (*nonfungiblePositionManagerAddress*)
- utils/DepositRecipes.sol - Line 41 - 47 (cache *tokenIds* outside of the loop)

Recommendation:
Consider caching the affected variables and reading from the cached version instead of doing costly read actions.

Update on the 14th of June 2022:
The developers have updated their implementation, taking our suggestion for gas improvements into consideration.

DONE - GO.6 - Save a storage slot by earlier cast

Location: actions/SwapToPositionRatio.sol - Lines 41 - 47

Description:
The pool variable is being obtained in two steps, with the outcome of the first step never being used again. To safe a storage slot and save gas, this can be optimized by directly casting the outcome of the first operation to IUniswapV3Pool.

Recommendation:
Consider directly casting the outcome of poolAddress to IUniswapV3Pool.

Update on the 14th of June 2022:
The developers have updated their implementation, taking our suggestion for gas improvements into consideration.

DONE - GO.7 - Unnecessary multiple approvals

Location: actions/ZapIn.sol - Lines 56

Description:
During the zapIn() call, the approveToken() function is called twice with the same inputs, hence wasting gas which is not necessary.

These are the affected lines of code:

```
ERC20Helper._approveToken(tokenIn, Storage.uniswapAddressHolder.swapRouterAddress(), amountIn);
[...]
ERC20Helper._approveToken(tokenIn, Storage.uniswapAddressHolder.swapRouterAddress(), amountIn);
```

Recommendation:
Consider removing one of the approval calls to save gas.

Update on the 14th of June 2022:
The developers have updated their implementation, taking our suggestion for gas improvements into consideration.

DONE - GO.8 - Possible caching opportunity

Location: modules/IdleLiquidityModule.sol - Lines 36 - 72

Description:
The result of the call to the NonfungiblePositionManager is already done in the _checkDistanceFromRange() function. It could be cached to prevent a double calling, saving some gas.

Recommendation:
Consider obtaining the required values from the _checkDistanceFromRange() function, caching them instead of calling the underlying function twice.

Update on the 14th of June 2022:
The developers have updated their implementation, taking our suggestion for gas improvements into consideration.

DONE - GO.9 - Possible caching opportunity

Location: Registry.sol - Lines 11

Description:
The access to the whitelistedKeepers array gets more and more expensive as the array size increases, with the cost of the access being $O(n)$. Making use of a hashmap (mapping) would bring this down to $O(1)$, which is way more efficient and cheaper.

Recommendation:
Consider switching from an array layout to a mapping/hashmap to make the access way more efficient (especially deletions).

Update on the 14th of June 2022:
The developers have updated their implementation, taking our suggestion for gas improvements into consideration.

3. Protocol/Logic Review

Part of our audits are also analyses of the protocol and its logic. The byterocket team went through the implementation and documentation of the implemented protocol.

The repository itself contained tests and documentation. We found the provided unit tests that are coming with the repository execute without any issues and cover the most important parts of the protocol.

According to our analysis, the protocol and logic are working as intended, given that any findings with a severity level are fixed. When making use of the Mainnet forking method, we were able to successfully execute the protocol and its modules.

We were **not able to discover any additional problems** in the protocol implemented in the smart contract.

4. Summary

During our code review (*which was done manually and automated*), we **found 39 bugs or flaws, with 39 of them being fixed** in a subsequent update**.** Prior to this, there have been 19 non-critical and 6 low, 9 medium and 5 high severity findings. Additionally, we found **9 gas improvements**, which have **all been implemented**.

The protocol review and analysis did neither uncover any game-theoretical nature problems nor any other functions prone to abuse besides the ones that have been uncovered in our findings.

In general, there are some improvements that can be made, but we are **very happy** with the overall quality of the code and its documentation. The developers have been very responsive and were able to answer any questions that we had.

Download the Report

Stored on IPFS

We store our public audit reports on IPFS; a peer-to-peer network called the "**I**nter **P**lanetary **F**ile **S**ystem". This allows us to store our reports in a distributed network instead of just a single server, so even if our website is down, every report is still available.

Learn more about IPFS →

Signed On-Chain

The IPFS Hash, a unique identifier of the report, is signed on-chain by both the client and us to prove that both sides have approved this audit report. This signing mechanism allows users to verify that neither side has faked or tampered with the audit.

Check the Signatures →