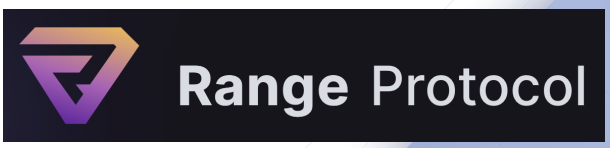
Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR



Range GHO Vault

Veridise Inc.

September 27, 2023

► **Prepared For:**

Range Protocol
https://www.rangeprotocol.com

► **Prepared By:**

Alberto Gonzalez
Benjamin Sepanski

► **Contact Us:** contact@veridise.com

► **Version History:**

Sep. 20, 2023 Initial Draft

Contents

1 Executive Summary

From Sep. 4, 2023 to Sep. 13, 2023, Range Protocol engaged Veridise to review the security of their Range GHO Vault. These vaults are designed to store user funds. A manager is responsible for taking out debt in GHO via an AAVE pool using user-supplied collateral, then managing the GHO and collateral in a Uniswap pool to produce profit in return for fees. The Veridise auditors reviewed the vault implementation and its associated factory, as well as slight modifications to Uniswap periphery contracts and renamings in an OpenZeppelin access-control contract.

The Veridise also simultaneously audited another repository supplied by the Range Protocol. This repository was the basis of the Range GHO Vault, and had nearly identical functionality. Most of the code was unchanged from this initial audit.

Veridise conducted the combined assessment over 2 person-weeks, with 2 engineers reviewing code over 1 week and 2 days on commits `0xef748c70-0xcaff8d88`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Range GHO Vault developers provided the source code of the Range GHO Vault contracts for review. To facilitate the Veridise auditors' understanding of the code, the developers provided online documentation explaining the high-level intent of the project. Developers also met with the Veridise team to give an in-depth walk-through of the code and point out areas of potential concern. The source code also contained some documentation in the form of READMEs and documentation comments on functions and storage variables.

The source code contained a test suite, which the Veridise auditors noted tested both positive and negative paths, verifying most of the access-control related paths. However, many of the tests from the prior code base had been removed or commented out, so the coverage was not as complete. Several files in the source code also indicate that the developers use linting such as prettier.

The Veridise auditors felt that the code was well organized and followed Solidity best practices. The vault had a very limited interface, which also reduces the attack surface. Note that this is under the assumption that the managers are a fully trusted entity, which was indicated to the auditors by the Range Protocol team.

During the audit, Range Protocol developers made 3 minor changes to the code base. The largest of these consisted of hard-coding the assumption that GHO's address is smaller than the address of the collateral token.

Summary of issues detected. The audit uncovered 18 issues, 3 of which are assessed to be of high severity by the Veridise auditors. These include incorrect use of decimals (V-RNGGHO-VUL-001 and V-RNGGHO-VUL-002), denial of service via underflows (V-RNGGHO-VUL-003).

The Veridise auditors also identified 2 medium-severity issues, including retroactive application of fees (V-RNGGHO-VUL-006) and lack of slippage protection (V-RNGGHO-VUL-017). The

audit uncovered a number of minor issues as well, including an out-of-bounds array access (V-RNGGHO-VUL-005), opportunities for small theft under mismanagement (V-RNGGHO-VUL-008), and several maintainability issues. The Range GHO Vault developers have resolved or acknowledged all 18 of these.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Range GHO Vault. Several of these recommendations are similar to recommendations given for the base protocol.

The auditors recommend to implement slippage protection as described in V-RNGGHO-VUL-005.

The auditors also recommend adding extensive documentation for managers. The Uniswap *core contracts are designed only to implement core functionality. Many issues (such as optimality and slippage protection) are implemented in the †periphery contracts. Managers may be prone to a host of mathematical or Solidity-specific errors handled in the periphery, and should be strongly recommended to rely on the heavily-used Uniswap contracts.

This guide should contain key management standards, as the manager is a trusted point in the protocol. It should also include the recommendation from issue V-RNGGHO-VUL-008.

The auditors also recommend creating a shared, base library to be used by both repositories during development. This would improve the auditability of both repositories and make the code easier to read about when switching from one repository to another.

Finally, the Veridise team recommends active monitoring of the protocol as an additional proactive defensive measure.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

* https://github.com/Uniswap/v3-core
† https://github.com/Uniswap/v3-periphery/

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Range GHO Vault	`0xef748c70` - `0xcaff8d8`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sep. 4 - Sep. 13, 2023	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	3	3
Medium-Severity Issues	2	2
Low-Severity Issues	3	3
Warning-Severity Issues	6	6
Informational-Severity Issues	4	4
TOTAL	18	18

Table 2.4: Category Breakdown.

Name	Number
Logic Error	6
Data Validation	6
Denial of Service	2
Maintainability	2
Transaction Ordering	1
Authorization	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Range GHO Vault's smart contracts. In our audit, we sought to answer the following questions:

- Is it possible to reenter the vault?
- Can reentry from the Uniswap pool lead to theft from the vault?
- Can manipulation of the Uniswap price between interactions with the vault enable theft?
- Can users profit from well-timed minting and burning?
- How can frontrunners profit from interactions with the vault?
- Can funds become locked in the vault?
- How can interactions with Aave affect the solvency of the vault?
- Are external APIs used correctly?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of our in-house static analyzer, Vanguard.

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy, flash loan attacks, uninitialized variables, and uses of variables before they are defined.

Scope. The scope of this audit is limited to the `contracts/` folder of the source code provided by the Range GHO Vault developers, which contains the smart contract implementation of the Range GHO Vault. Namely, the `RangeProtocolFactory.sol`, `RangeProtocolVault.sol`, and `RangeProtocolStorage.sol` files, the core implementation inside `libraries/LogicLib.sol` and `libraries/DataTypesLib.sol`, along with the `access/`, `errors/`, and `uniswap/` directories.

Methodology. Veridise auditors reviewed the reports of previous audits for Range GHO Vault, inspected the provided tests, and read the Range GHO Vault documentation. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Range GHO Vault developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-RNGGHO-VUI	Incorrect computation of the Performance Fee	High	Fixed
V-RNGGHO-VUI	The value of priceFromOracle is computed incorr.	High	Fixed
V-RNGGHO-VUI	Underflow reverts before cast to int256 in the . . .	High	Fixed
V-RNGGHO-VUI	Retroactive fees	Medium	Fixed
V-RNGGHO-VUI	No slippage protection in mint(), swap(), or bu. . .	Medium	Fixed
V-RNGGHO-VUI	Manager fee is not included in deficit computation	Low	Fixed
V-RNGGHO-VUI	Out-of-bounds array access	Low	Fixed
V-RNGGHO-VUI	Poor management of fees may lead to small theft	Low	Acknowledged
V-RNGGHO-VUI	Should require token0 is gho	Warning	Fixed
V-RNGGHO-VUI	Ensure at least one token in pool is _gho	Warning	Fixed
V-RNGGHO-VUI	Existing issues from prior reports	Warning	Acknowledged
V-RNGGHO-VUI	Verify decimals from chainlink oracle	Warning	Fixed
V-RNGGHO-VUI	Manager address could be zero	Warning	Fixed
V-RNGGHO-VUI	Inconsistent behavior in updateTicks()	Warning	Fixed
V-RNGGHO-VUI	Shares should round down	Info	Fixed
V-RNGGHO-VUI	Recommended Monitoring: Liquidity share in poo	Info	Acknowledged
V-RNGGHO-VUI	Should use helper function	Info	Fixed
V-RNGGHO-VUI	Recommended Invariant: Uniswap pool is unlocke	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-RNGGHO-VUL-001: Incorrect computation of the Performance Fee

Severity	High	Commit	caff8d8
Type	Logic Error	Status	Fixed
File(s)	libraries/LogicLib.sol		
Location(s)	_applyPerformanceFee()		
Confirmed Fix At	09ff644		

The `_applyPerformanceFee` function in the `LogicLib` file is used to update the `managerBalance` from the fees collected in the Uniswap Pool.

The issue is that `managerBalance` is tracked in `collateral` token units but the fees in `gho` are not converted to `collateral` units. The computation is done as shown in the below snippet.

```
state.managerBalance += (fee0 * uint256(ghoPrice) * _performanceFee) /
    uint256(collateralPrice) / 10_000;
```

Analyzing the above computation we have:

- `fee0` is in terms of `gho`.
- `ghoPrice / collateralPrice` returns `collateral / gho`.
- `_performanceFee / 10000` it is the percentage computation.

Impact The `managerFee` will be much larger for the `USDC` vault, for example, since `gho` has 18 decimals and `USDC` has 6.

Recommendation Divide by the `gho` decimals from `fee0` and multiply by the `collateral` decimals.

Developer Response The issue is acknowledged and fixed.

4.1.2 V-RNGGHO-VUL-002: The value of priceFromOracle is computed incorrectly

Severity	High	**Commit**	ef748c7
Type	Denial of Service	**Status**	Fixed
File(s)	`libraries/LogicLib.sol`		
Location(s)	_isPriceWithinThreshold()		
Confirmed Fix At	caff8d8		

The `_isPriceWithinThreshold` function in the `LogicLib` file is used to validate that the price of the Uniswap Pool is within a threshold of the price returned by Chainlink Oracles. This is to avoid consuming a manipulated price. The logic within this function computes a variable named `priceFromOracle` using the fetched prices `collateralPrice` and `ghoPrice`:

```
1 uint256 priceFromOracle = state.isToken0GHO
2     ? (10 ** state.decimals1 * uint256(ghoPrice)) / uint256(collateralPrice)
3     // @audit-issue It should multiply here by 10**state.decimals1 instead of 10**
    state.decimals0.
4     : (10 ** state.decimals0 * uint256(collateralPrice)) / uint256(ghoPrice);
```

Snippet 4.1: Computation of the variable `priceFromOracle` in the `_isPriceWithinThreshold` function.

The issue is in the computation of `priceFromOracle` during the `else` branch, where GHO is not token 0.

The code multiplies `collateralPrice` by `state.decimals0`, but if we follow the consistency of the formula units, we expect from `priceFromOracle` to have the units of `token1 / token0`. Hence, the formula should use `10 ** state.decimals1`.

Impact The value of `priceFromOracle` will be represented in the amount of decimals of `token0` instead of `token1`.Iif those are different, then the threshold validation may fail.

Recommendation Change the `else` from:

```
: (10 ** state.decimals0 * uint256(collateralPrice)) / uint256(ghoPrice);
```

to

```
: (10 ** state.decimals1 * uint256(collateralPrice)) / uint256(ghoPrice);
```

Developer Response The issue is acknowledged. We have removed the else branch since our GHO vaults will only be created with the GHO pair with USDC, DAI and LUSD whose integer representation is greater than GHO's address this which results in GHO always being token0 in those vaults.

4.1.3 V-RNGGHO-VUL-003: Underflow reverts before cast to int256 in the computation of gho deficit

Severity	High	**Commit**	ef748c7
Type	Denial of Service	**Status**	Fixed
File(s)	`libraries/LogicLib.sol`		
Location(s)	getBalanceInCollateralToken()		
Confirmed Fix At	5d46309		

The `getBalanceInCollateralToken()` function in the `LogicLib` file is used to track all the balances of `gho` and `collateral` that the vault has and then convert them into a single value expressed in the `collateral` token.

The branch where `gho` is `token1` will lead us to the following code:

```
vars.token0BalanceSigned =
          int256(vars.amount0FromPool + state.token0.balanceOf(address(this))) +
          int256(vars.amount0FromAave);
vars.token1BalanceSigned = int256(
          vars.amount1FromPool + state.token1.balanceOf(address(this)) - vars.
    amount1FromAave);
```

Snippet 4.2: Logic found in the `getBalanceInCollateralToken` function of the `LogicLib`.

The issue is in the computation of `vars.token1BalanceSigned`, which is a value that could be negative in the case the vault has a deficit of `gho`. However, the code will always revert when the deficit exists, since: `vars.amount1FromPool + state.token1.balanceOf(address(this)) - vars.amount1FromAave);` is an `uint256` value that cannot underflow.

Impact For vaults that have the `gho` token as `token1`, a `gho` deficit will lead to denial of service.

Recommendation Rewrite it as:

```
vars.token1BalanceSigned = int256(vars.amount1FromPool + state.token1.balanceOf(address
(this)) - int256(vars.amount1FromAave);
```

Developer Response The issue is acknowledged. We have made changes in the code as we will pair GHO with USDC, DAI and LUSD where token1 will always be non-GHO token and is only added to the "`vars.token1Balance`" removing the possibility of underflow.

4.1.4 V-RNGGHO-VUL-004: Retroactive fees

Severity	Medium	**Commit**	ef748c7
Type	Logic Error	**Status**	Fixed
File(s)	`libraries/LogicLib.sol`		
Location(s)	updateFees()		
Confirmed Fix At	09ff644		

The `updateFees()` function allows the manager to change the fees.

```
function updateFees(
    uint16 newManagingFee,
    uint16 newPerformanceFee
) external override onlyManager {
    if (newManagingFee > MAX_MANAGING_FEE_BPS) revert VaultErrors.InvalidManagingFee
    ();
    if (newPerformanceFee > MAX_PERFORMANCE_FEE_BPS) revert VaultErrors.
    InvalidPerformanceFee();

    managingFee = newManagingFee;
    performanceFee = newPerformanceFee;
    emit FeesUpdated(newManagingFee, newPerformanceFee);
}
```

Snippet 4.3: Implementation of `updateFees()`.

However, performance fees are not collected until fees are pulled, tokens are burned, or liquidity is removed. Thus, the new performance fee will be applied retroactively on fees garnered from liquidity in the uniswap pool.

Impact Users may be forced to pay larger fees than indicated in the contract.

Recommendation Collect uniswap fees from the pool and apply the old performance fee before updating to the new fee.

Developer Response Issue acknowledged and fix is applied.

4.1.5 V-RNGGHO-VUL-005: No slippage protection in mint(), swap(), or burn()

Severity	Medium	Commit	ef748c7
Type	Transaction Ordering	Status	Fixed
File(s)	RangeProtocolVault.sol		
Location(s)	mint(), burn()		
Confirmed Fix At	83448cf		

Callers of `mint()` submit a `mintAmount`, and then the `LogicLib` computes how much of each token is owed. For example, the `totalSupply > 0` case is shown in the below code snippet.

```
(uint256 amount0Current, uint256 amount1Current) = getUnderlyingBalances();
amount0 = FullMath.mulDivRoundingUp(amount0Current, mintAmount, totalSupply);
amount1 = FullMath.mulDivRoundingUp(amount1Current, mintAmount, totalSupply);
```

Snippet 4.4: Computation of `(amount0, amount1)` when `totalSupply > 0` in `mint()`.

Once this is done, the specified amounts of each token is transferred to the pool.

```
if (amount0 > 0) {
    userVaults[msg.sender].token0 += amount0;
    token0.safeTransferFrom(msg.sender, address(this), amount0);
}
if (amount1 > 0) {
    userVaults[msg.sender].token1 += amount1;
    token1.safeTransferFrom(msg.sender, address(this), amount1);
}
```

Snippet 4.5: Transfer of funds inside `mint()`.

The amounts are never checked, and may be larger than the user expects. In order to protect against this, a user must have approved only a small amount.

Similarly, `burn()`, `swap()`, `addLiquidity()`, and `removeLiquidity()` have no slippage protection.

Impact Users have the largest potential impact with `mint()` and `burn()`. If a large `swap()` occurs on the Uniswap pool right before the `mint()` (resp. `burn()`), the value expended may become much larger (resp. smaller) than expected. For `mint()`, a user who naively approves `uint.max` for the pool may then pay more than they wish. For `burn()`, regardless of user action they may receive less than expected.

Swapping and adding/removing liquidity primarily affects the manager, who must be aware of the possibility that their action could be frontrun.

Recommendation We recommend adding a parameter to provide slippage protection.

If not possible, we suggest noting in the documentation that a user must approve only the amount they wish to spend for `mint()`, but still adding slippage protection for `burn()`. The developers should also document that the managers are expected to check for slippage.

Developer Response It seems to be only applicable to main vault since GHO vault does not interact with AMM pool in mint and burn functions.

Veridise Response AMM interaction is not the only reason why slippage might be necessary. Any tokenized vault is encouraged to add slippage protection as per ERC-4626. For example, your transaction may get stuck. Meanwhile, the protocol can earn fees (increasing the price of shares) or change the range.

For more information, please review the Security Considerations https://eips.ethereum.org/EIPS/eip-4626.

While slippage for EOAs is just a recommendation and not mandatory per the speck, we still recommend adding it in.

Updated Developer Response Slippage protection has been added.

4.1.6 V-RNGGHO-VUL-006: Manager fee is not included in deficit computation

Severity	Low	**Commit**	ef748c7
Type	Maintainability	**Status**	Fixed
File(s)	`libraries/LogicLib.sol`		
Location(s)	getBalanceInCollateralToken()		
Confirmed Fix At	09ff644		

The `getBalanceInCollateralToken()` function in the `LogicLib` file is used to track all the balances of `gho` and `collateral` that the vault has and then convert them into a single value expressed in the `collateral` token.

Inside this logic, the code computes if the vault has a deficit of `gho` token, which might happen due to users buying `gho` from the Uniswap Pool minus the `gho` debt in AAVE. When the vault indeed has a deficit of `gho`, it converts that deficit to an equivalent amount of `collateral` token, which gets deducted from the returned amount of the function `getBalanceInCollateralToken`.

The `managerFee` is not used in the computation of the `gho` deficit which might lead the function to return an incorrect amount of `collateral` token.

Impact Take for example the following scenario:

- `gho` token balance = 100 gho-tokens
- `managerFee` balance = 105 gho-tokens

Because the `gho` balance is greater than zero, the vault will think that it does not have a deficit. However, due to the fact that `managerFee > gho`, the `gho` balance will be set to zero but the `collateral` amount will not get decremented.

Recommendation Include the `managerFee` in the computation of the `gho` deficit.

Developer Response We did a revamp to the logic and took into account the manager balance subtraction at the total holding of vault in collateral token.

4.1.7 V-RNGGHO-VUL-007: Out-of-bounds array access

Severity	Low	**Commit**	ef748c7
Type	Logic Error	**Status**	Fixed
File(s)	`RangeProtocolFactory.sol`		
Location(s)	getVaultAddresses()		
Confirmed Fix At	09ff644		

The function `getVaultAddresses()` in `RangeProtocolFactory.sol` allows callers to receive any contiguous subset of the `_vaultsList` from a `startIdx` through an `endIdx` by copying the values into a local `vaultList` array, and returning that to the user.

```
function getVaultAddresses(
        uint256 startIdx,
        uint256 endIdx
    ) external view returns (address[] memory vaultList) {
        vaultList = new address[](endIdx - startIdx + 1);
        for (uint256 i = startIdx; i <= endIdx; i++) {
            vaultList[i] = _vaultsList[i];
        }
    }
```

Snippet 4.6: Implementation of `getVaultAddresses()`

However, the `vaultList` is indexed at `i`, which ranges from `startIdx` through `endIdx`, rather than at `i-startIdx`, which ranges from `0` through `endIdx-startIdx`.

Impact Any call to `getVaultAddresses()` with `startIdx > 0` will cause an out-of-bounds array access in the last few iterations of the loop.

Recommendation Replace `vaultList[i] = _vaultsList[i];` with `vaultList[i-startIdx] = _vaultsList[i];`.

Developer Response The issue is acknowledged and a fix is applied.

4.1.8 V-RNGGHO-VUL-008: Poor management of fees may lead to small theft

Severity	Low	**Commit**	ef748c7
Type	Logic Error	**Status**	Acknowledged
File(s)	`libraries/LogicLib.sol`		
Location(s)	getBalanceInCollateralToken()		
Confirmed Fix At	N/A		

When calling `getBalanceInCollateralToken()`, the manager balance is subtracted from the accrued fees if the accrued fees are large enough.

```
(uint256 burn0, uint256 burn1, uint256 fee0, uint256 fee1) = _withdraw(
    liquidityBurned);

_applyPerformanceFee(fee0, fee1);
(fee0, fee1) = _netPerformanceFees(fee0, fee1);
emit FeesEarned(fee0, fee1);

uint256 passiveBalance0 = token0.balanceOf(address(this)) - burn0;
uint256 passiveBalance1 = token1.balanceOf(address(this)) - burn1;
if (passiveBalance0 > managerBalance0) passiveBalance0 -= managerBalance0;
if (passiveBalance1 > managerBalance1) passiveBalance1 -= managerBalance1;

amount0 = burn0 + FullMath.mulDiv(passiveBalance0, burnAmount, totalSupply);
amount1 = burn1 + FullMath.mulDiv(passiveBalance1, burnAmount, totalSupply);
```

Snippet 4.7: `inThePosition` case of `burn()`.

This means that, if the manager balance is very near, but below, the passive balance, a user who burns from the vault will receive more than expected.

Impact If managers keep enough liquid values to just barely cover their fees, they may be vulnerable to small thefts from the vault.

The following example is performed on a similar vault with nearly identical logic from the Range team, which is not restricted to GHO.

For example, the below scenario shows a situation where

1. An attacker mints in the vault.
2. The manager pulls fees from the pool then rebalances liquidity leaving just enough for fees.
3. The attacker burns their tokens, profiting from the exchange.

```
Current fees from pool: (0.200229,0.200004)
Attacker quickly mints before fees are pulled from the pool
Cost: (1.994866,1.994872), Tokens Minted: 2000000000000000000
Fees are pulled from pool
Vault balance:   (0.971597,1.136293)
Manager balance: (0.005134,0.005128)
Manager adds liquidity to the pool, leaving just enough for fees
Vault balance:   (0.005134,0.222477)
Manager balance: (0.005134,0.005128)
```

```
Attacker burns all tokens!
Attacker reward: (1.997433,1.994872)
Attacker profit: (0.002567,-0.000000)
```

Note that the amounts stolen here are typically small.

Recommendation We recommend either reverting when the passive balance cannot cover the debt to the manager, or at least emitting an event.

If choosing not to revert, the range protocol should be sure to inform managers of this possibility, and ensure the managers withdraw fees frequently.

Developer Response The off-chain strategy of managers will take the decision of either to keep the manager fee in the vault or deploy it on the pool when adding liquidity along with the assets.

4.1.9 V-RNGGHO-VUL-009: Should require token0 is gho

Severity	Warning	Commit	caff8d8
Type	Data Validation	Status	Fixed
File(s)	RangeProtocolVault.sol		
Location(s)	initializer()		
Confirmed Fix At	09ff644		

As the new implementation assumes that `token0` is `_gho`, this should be `required` in the `initializer()`.

Impact Future deployments of the vault may fail unexpectedly.

Recommendation Require that `token0` is `_gho`.

Developer Response The issue is acknowledged and a fix is applied.

4.1.10 V-RNGGHO-VUL-010: Ensure at least one token in pool is _gho

Severity	Warning	Commit	ef748c7
Type	Data Validation	Status	Fixed
File(s)	`RangeProtocolVault.sol`		
Location(s)	initialize()		
Confirmed Fix At	09ff644		

The `pool` address received by the `initializer()` has two tokens. One is assumed to be `_gho`, as shown in the below logic.

```
if (address(token0) == _gho) {
    state.isToken0GHO = true;
    state.vaultDecimals = decimals1;
} else {
    state.vaultDecimals = decimals0;
}
```

Snippet 4.8: Determination of `isToken0GHO` in `initializer()`.

Impact If the wrong pool is supplied, then neither token may be equal to `_gho`, causing other errors throughout the contract.

Recommendation Require that `address(token1) == _gho` in the `else {` branch during initialization.

Developer Response ****Ensures token 0 is gho.

4.1.11 V-RNGGHO-VUL-011: Existing issues from prior reports

Severity	Warning	**Commit**	caff8d8
Type	Authorization	**Status**	Acknowledged
File(s)		N/A	
Location(s)		N/A	
Confirmed Fix At		N/A	

Here we highlight several issues identified by previous audit reports which remain open, and add our encouragement to that of prior auditors for the developers to address these issues.

1. **HAL-04: Users can steal any manually added liquidity.** In both repositories, if a manager adds liquidity manually, then vault tokens become more valuable and attackers may frontrun to buy the tokens while cheap, then burn them once they've gained value.
2. **HAL-05: Fee payment bypass is possible for small amounts.** Since fees are computed with only 4 decimals, small burn amounts may avoid fees.
3. **HAL-07: Malicious manager can steal a share of vault liquidity.** In both repositories, the managers are trusted entities. For instance, a manager may perform a large number of swaps with the users own funds simply to generate fees.

Impact

1. Manually added liquidity may go to waste.
2. Small fees may go unpaid.
3. Untrusted managers, or managers who are hacked, may use bad swaps to steal user funds.

Recommendation

1. Be sure to document this possibility and make it clear that managers should not add liquidity manually.
2. Notify managers of this possibility, or set a minimum fee.
3. Make clear to users that managers must be fully trusted, and add extra requirements on managers to ensure their keys are stored safely.

Developer Response Fix is applied by using decimals from oracle price feed.

4.1.12 V-RNGGHO-VUL-012: Verify decimals from chainlink oracle

Severity	Warning	Commit	ef748c7
Type	Data Validation	Status	Fixed
File(s)	`RangeProtocolVault.sol`		
Location(s)	initialize()		
Confirmed Fix At	6f8ebed		

The computation to `getUnderlyingBalancesFromAave()` uses prices for GHO and the collateral token supplied by chainlink. It uses these prices to convert from the Aave's user account data (which is provided in units of the base currency) to token amounts.

```
(uint256 totalCollateralBase, uint256 totalDebtBase, , , , ) = IPool(state.
    poolAddressesProvider.getPool())
    .getUserAccountData(address(this));

(amount0, amount1) = state.isToken0GHO
    ? (
        (totalDebtBase * 10 ** state.decimals0) / ghoPrice,
        (totalCollateralBase * 10 ** state.decimals1) / collateralTokenPrice
    )
    : (
        (totalCollateralBase * 10 ** state.decimals0) / collateralTokenPrice,
        (totalDebtBase * 10 ** state.decimals1) / ghoPrice
);
```

Snippet 4.9: Definition of `getUnderlyingBalancesFromAave()`.

This assumes that the number of decimals in the reported price, and the base currency of the Aave pool, match the reported currency of the chainlink oracle.

The Aave docs state that

> All V3 markets use USD based oracles which return values with 8 decimals.

While the chainlink oracles currently intended to be used do report the price in USD with 8 decimals, future deployments of this protocol (or updates to Aave or Chainlink) may cause severe issues.

Impact While unlikely, a mismatch in the currency which Chainlink oracles report in and the currency which Aave pools report in could cause severe miscalculations when computing underlying balances.

Recommendation Verify that the chainlink oracles use 8 decimals during initialization of the `RangeProtocolVault`. Make note in the documentation of this assumption, and be sure to follow updates to the Aave pool and Chainlink oracles used.

Developer Response We have taken the AAVE and chainlink decimals into account in our computations.

4.1.13 V-RNGGHO-VUL-013: Manager address could be zero

Severity	Warning	Commit	ef748c7
Type	Data Validation	Status	Fixed
File(s)	RangeProtocolVault.sol		
Location(s)	initialize()		
Confirmed Fix At	09ff644		

The `manager` for the vault is provided through the `data` parameter of `initialize()`. Then, ownership is transferred to the `manager`.

```
function initialize(
    address _pool,
    int24 _tickSpacing,
    bytes memory data
) external override initializer {
    (address manager, string memory _name, string memory _symbol) = abi.decode(
        data,
        (address, string, string)
    );

    __UUPSUpgradeable_init();
    __ReentrancyGuard_init();
    __Ownable_init();
    __ERC20_init(_name, _symbol);
    __Pausable_init();

    _transferOwnership(manager);
```

Snippet 4.10: Implementation of `initialize()`.

The caller of initialize is `_createVault()` inside `RangeProtocolFactory`:

```
function _createVault(
    address tokenA,
    address tokenB,
    uint24 fee,
    address pool,
    address implementation,
    bytes memory data
) internal returns (address vault) {
    if (data.length == 0) revert FactoryErrors.NoVaultInitDataProvided();
    // .... irrelevant code elided
    vault = address(
        new ERC1967Proxy(
            implementation,
            abi.encodeWithSelector(INIT_SELECTOR, pool, tickSpacing, data)
        )
    );
    _vaultsList.push(vault);
}
```

Snippet 4.11: Implementation of `_createVault()`.

So, the only validation performed on `data` is length-based.

Impact The `manager` address may be zero without causing any errors during initialization.

Recommendation Revert during initialization if the `manager` address (or other user-supplied addresses) are `0x0`.

Developer Response Fix is applied.

4.1.14 V-RNGGHO-VUL-014: Inconsistent behavior in updateTicks()

Severity	Warning	**Commit**	ef748c7
Type	Logic Error	**Status**	Fixed
File(s)	`libraries/LogicLib.sol`		
Location(s)	updateTicks()		
Confirmed Fix At	09ff644		

The `updateTicks()` function in the contracts repository sets `inThePosition` to `true`, preventing calls to `addLiquidity()` until the liquidity is removed.

In the Range-GHO-Vault repository, `updateTicks()` does not set `inThePosition` to `true`. This is likely due to the fact that minting does not directly deposit into the Uniswap pool, as in the non-GHO vault.

Impact This behavior may be confusing to users of the protocol. In particular, a user may `mint()` after a call to `updateTicks()`, expecting to see a `LiquidityRemoved` event before the ticks change. However, the manager may call `addLiquidity()` immediately after and change the ticks to values without triggering the `LiquidityRemoved` event.

Recommendation Remove `updateTicks()`, or make its behavior the same in both types of vaults. Alternatively, document the difference in behavior.

Developer Response `updateTicks()` function is removed.

4.1.15 V-RNGGHO-VUL-015: Shares should round down

Severity	Info	**Commit**	ef748c7
Type	Logic Error	**Status**	Fixed
File(s)	`libraries/LogicLib.sol`		
Location(s)	mint()		
Confirmed Fix At	1a10012		

During the computation of `shares` to mint during the `mint` function of the `LogicLib` file, the logic rounds up the value:

```
1 shares = FullMath.mulDivRoundingUp(amount, totalSupply, totalAmount);
```

Snippet 4.12: Logic from the `mint` function in the `LogicLib` file used to calculate the amount of `shares` to mint.

This means that the logic rounds in favor of the user instead of the protocol. Per the standard `ERC4626` that can be found here, tokenized vaults should round down when computing the `shares` value.

Impact It is important to notice that in the current implementation, rounding up prevents the so called "Inflation Attack". The attack is prevented due to the fact that the returned `shares` can never be zero, since by rounding up, you ensure at least `1` share minted.

Recommendation We recommend documenting this deviation from `ERC4626` so it is made explicit in the specification of the protocol.

If the team does not want a deviation from `ERC4626`, then special attention needs to be taken in regard with the inflation attack. More detail about the mitigations of this attack can be found here:

https://blog.openzeppelin.com/a-novel-defense-against-erc4626-inflation-attacks

Developer Response Documented the behavior of rounding up in the code.

4.1.16 V-RNGGHO-VUL-016: Recommended Monitoring: Liquidity share in pool

Severity	Info	**Commit**	ef748c7
Type	Data Validation	**Status**	Acknowledged
File(s)		N/A	
Location(s)		N/A	
Confirmed Fix At		N/A	

The Range-GHO-Vault has a circuit breaker when the uniswap pool price deviates from the chainlink price by more than 0.5%. By our computations, for a position which is spread across the entire tick range at the lowest Uniswap fee tier of 0.05%, a price manipulation attack on the vault (i.e. to inflate token value before burning) may be profitable when the vault's share of pool liquidity reaches around 40%.

This is a rough estimate based on many assumptions, including that the assets in the pool have roughly the same price.

We recommend the developers monitor the vault's share of pool ownership. We also recommend they limit the manager so that they cannot accidentally add enough liquidity to make up more than 10% of the pool. This could be implemented as an override on `addLiquidity()`, which checks that the vault does not own too much of the pool unless the manager explicitly indicates this is intentional.

4.1.17 V-RNGGHO-VUL-017: Should use helper function

Severity	Info	Commit	ef748c7
Type	Maintainability	Status	Fixed
File(s)	`libraries/LogicLib.sol`		
Location(s)	getUnderlyingBalancesFromAave()		
Confirmed Fix At	09ff644		

The function `getUnderlyingBalancesFromAave()` gets the user account data from the Aave pool.

```
function getUnderlyingBalancesFromAave(
    DataTypesLib.State storage state
) public view returns (uint256 amount0, uint256 amount1) {
    (uint256 totalCollateralBase, uint256 totalDebtBase, , , , ) = IPool(state.
    poolAddressesProvider.getPool())
        .getUserAccountData(address(this));
```

Snippet 4.13: Snippet from `getUnderlyingBalancesFromAave()`.

This first line is exactly the definition of `getAavePositionData()`.

```
function getAavePositionData(
    DataTypesLib.State storage state
)
    external
    view
    returns (
        uint256 totalCollateralBase,
        uint256 totalDebtBase,
        uint256 availableBorrowsBase,
        uint256 currentLiquidationThreshold,
        uint256 ltv,
        uint256 healthFactor
    )
{
    return IPool(state.poolAddressesProvider.getPool()).getUserAccountData(address(
    this));
}
```

Snippet 4.14: Definition of `getAavePositionData()`.

Impact If the definition of the helper function changes, developers must remember to make the change in both places.

Recommendation Use the helper function instead of its inlined implementation.

Developer Response Fix is applied.

4.1.18 V-RNGGHO-VUL-018: Recommended Invariant: Uniswap pool is unlocked

Severity	Info	Commit	ef748c7
Type	Data Validation	Status	Acknowledged
File(s)	`libraries/LogicLib.sol, RangeProtocolVault.sol`		
Location(s)	mint(), burn()		
Confirmed Fix At	N/A		

The functions `mint()` and `burn()` in the `LogicLib` do not call any Uniswap functions with the `lock` modifier.

We recommend checking that the Uniswap pool is currently unlocked to reduce the attack surface for price manipulation. This can be asserted using the slot0() function.

Developer Response ****Manager has the role to pause the minting and burning in the event the AMM pool is manipulated.

Glossary

AAVE Aave is an Open Source Protocol to create Non-Custodial Liquidity Markets to earn interest on supplying and borrowing assets. To learn more, visit https://aave.com . 1

AMM Automated Market Maker. 29

GHO A stablecoin native to the AAVE protocol. To learn more, visit https://docs.aave.com/faq/gho-stablecoin. 1

OpenZeppelin A security company which provides many standard implementations of common contract specifications. See `https://www.openzeppelin.com`. 1

prettier A code formatting tool, see `https://prettier.io/docs/en/integrating-with-linters.html` to learn more. 1

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure.. 29

Solidity The standard high-level language used to develop smart contracts on the Ethereum blockchain. See `https://docs.soliditylang.org/en/v0.8.19/` to learn more. 2

Uniswap One of the most famous deployed AMMs. See `https://uniswap.org` to learn more. 1