QuillAudits

Audit Report
November, 2022

For

UniFarm

Table of Content

Table of Content

Executive Summary

Project Name	Unifarm Bridge (Setu bridge)
Overview	This is a bridge project that utilizes Router protocol for cross chain transfer of assets. There are two types of bridges- liquidity bridge and child bridge used in this project. Setu token has been used to facilitate the crosschain transfer and tracking of the assets. Transparent Upgradeable contracts from Openzeppelin have been used in designing this project.
Timeline	10th August, 2022 to 14th october, 2022
Method	Manual Review, Functional Testing, Automated Testing, etc.
Scope of Audit	The scope of this audit was to analyze Unifarm codebase for quality, security, and correctness *https://github.com/myunifarm/unifarm-bridge/tree/dev-2/contracts*
Commit Hash	The dev-2 branch of this repo was provided for the audit at commit f92ed73643c376bcb7e0d647287641ad71cd2e96
Fixed In	*https://github.com/myunifarm/unifarm-bridge/commit/4afa69ea07af33c2466012dd24ce2b97c7472a44*

High Medium Low Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	3	2	3
Partially Resolved Issues	0	0	1	0
Resolved Issues	1	11	13	1

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - TokenUpgradeable

High Severity Issues

No issues found

Medium Severity Issues

A.1 Centralization of burnTokens() allows burning tokens from any account

Description

burnTokens() function can be called by the owner to burn tokens from any account, even from the accounts that it currently does not own. Here the owner is a bridge contract, so this intended design is fine.

But this can be exploited by a malicious admin if he calls renounceOwnership() and transfer ownership of the contract to himself. Also lockedTokens() function can be called by the owner to burn tokens from any account, even from the accounts that it currently does not own.

Also a malicious owner can mint more setuTokens and use it to drain liquidity from the Liquidity Pools by calling the transferOut() function.

```
45      function burnTokens(address from, uint256 amount) public onlyOwner {
46          _burn(from, amount);
47      }
48
```

Remediation

It would be much safer if the transition is managed by implementing a two-step approach: _transferOwnership() and _updateOwnership() . Specifically, the _transferOwnership () function keeps the new address in the storage, _newOwner , instead of modifying the _owner() directly. The updateOwnership() function checks whether _newOwner is msg.sender, which means _newOwner signs the transaction and verifies himself as the new owner. After that, _newOwner could be set into _owner.

Status

Resolved

Comment: The unifarm team said that the owner of the contract will be gnosis multisig contract

A.2 Missing usage of pausable modifier

Description

Pausable modifier from the Openzeppelin is not used. Calling pauseTokens() does not actually pause any functionality in the contract like locktokens() or unlocktokens().

Remediation

Use the pausable modifier from the inherited PausableUpgradeable contract where necessary.

```
53      function pauseTokens() public onlyOwner {
54          _pause();
55      }
56
57      function unpauseTokens() public onlyOwner {
58          _unpause();
59      }
```

Status

Resolved

Low Severity Issues

No issues found

Informational Issues

A.3 Missing zero address check

Description

Missing zero address check in for _admin parameter in initialize() function.

Remediation

It is advised to add a require check for the same.

```
24      function initialize(
25          string calldata _name,
26          string calldata _ticker,
27          uint8 _decimal,
28          address _admin
29      ) public initializer {
30          __ERC20_init(_name, _ticker);
31          // __ERC20Burnable_init();
32          __Ownable_init();
33          admin = _admin;
34          _decimals = _decimal;
35      }
```

Status

Resolved

B. Contract - TokenBridgeRegistryUpgradeable

High Severity Issues

No issues found

Medium Severity Issues

B.1 No sanity checks for bridgeType in addBridge()

Description

There are no sanity checks for bridgeType in addBridge() function. According to the docs provided there can be a maximum of 2 bridge types- liquidity bridge and child bridge (or liquidity + child bridge). BridgeUpgradeable contract's transferIn() which has if statements depending on bridgeType, will not get executed if the bridgeType is incorrectly set other than value 0 or 1. Also this could lead to the possibility that the crossChainTransferIn() is done successfully without paying any fees (plus the user gets additional setutokens minted which would actually should have been burned).

```
116     function addBridge(
117         uint8 _bridgeType,
118         string memory _tokenTicker,
119         uint256 _epochLength,
120         uint8 _feeType,
121         uint256 _feeInBips
122     ) public onlyOwner {
123         require(bridgeTokenMetadata[_tokenTicker].tokenAddress != address(0), "TOKEN_NOT_EXISTS");
124         require(tokenBridge[_tokenTicker].startBlock == 0, "TOKEN_BRIDGE_ALREADY_EXISTS");
125         require(_feeType == 0 || _feeType == 1, "INVALID_FEE_TYPE");
```

Remediation

Add sufficient require checks for bridgeType the same in addBridge()

Status

Resolved

B.2 Centralization of updateFeeConfig() and updateEpochLength()

Description

updateFeeConfig() allows updating of tokenBridge's fee at any time. Also the _feeInBips parameter has no upper limit and can be exploited by a malicious admin to set very high fees without anyone noticing or taking action. Also
_feeInBips parameter lacks zero value check as well which can lead to unintended issues such as no fees getting paid by users.

updateEpochLength() allows to update the epochLength of bridge anytime which can lead to unintended issues and bugs.

```
195     function updateFeeConfig(
196         string calldata _tokenTicker,
197         uint8 _feeType,
198         uint256 _feeInBips
199     ) public onlyOwner {
200         require(tokenBridge[_tokenTicker].startBlock > 0, "NO_TOKEN_BRIDGE_EXISTS");
201         if(_feeType == 1) {
202             _feeInBips *= 100;
203         }
204         FeeConfig memory feeConfig = FeeConfig({
205             feeType: _feeType,
206             feeInBips: _feeInBips
207         });
208         tokenBridge[_tokenTicker].fee = feeConfig;
209     }
210
211     function updateEpochLength(
212         string calldata _tokenTicker,
213         uint256 _newEpochLength
214     ) public onlyOwner {
215         require(tokenBridge[_tokenTicker].startBlock > 0, "NO_TOKEN_BRIDGE_EXISTS");
216         tokenBridge[_tokenTicker].epochLength = _newEpochLength;
217     }
```

Remediation

It is advised to add sufficient require checks for _feeInBips in the function. A multisig with at least ¾ configuration is advised to be the owner of the contract. Also it is recommended to allow epochLength updation only after existing fees have been claimed by the LPs.

Status

Resolved

Comment: For updateFeeConfig(), added check for totalFees to be 0. The unifarm team said that updateEpochLength() would not have any issue as they are storing epoch length for each passed epoch in the epochs array.

B.3 Centralization of addTokenMetadata()

Description

A malicious owner can deploy a dummy or fake setu token and overwrite the existing metadata of already added setu token using addTokenMetadata using addTokenMetadata() function.

For example, let's say setuUSDC was deployed using addBridge() for USDC token, then a malicious owner can overwrite the bridgeTokenMetadata mapping for the USDC setuToken. This is possible if the owner had deployed an ERC20 token with symbol as setuUSDC and the address of this token is passed as a parameter to the addTokenMetadata() function.

This attack can be coupled with A.1 to amplify this attack.
In addition to this, overwriting to bridgeTokenMetadata can also be done via _deploySetuToken if a setuToken with the same ticker and symbol is deployed again by an owner.

```
98      function addTokenMetadata(
99          address _tokenAddress,
100         string calldata _imageUrl
101     ) public onlyOwner {
102         TokenUpgradeable token = TokenUpgradeable(_tokenAddress);
103
104         BridgeTokenMetadata memory newBridgeTokenMetadata = BridgeTokenMetadata ({
105             name: token.name(),
106             imageUrl: _imageUrl,
107             tokenAddress: _tokenAddress
108         });
109         bridgeTokenMetadata[token.symbol()] = newBridgeTokenMetadata;
110     }
```

Remediation

Use multisig for admin address to increase decentralization and minimize the risk of malicious use of addTokenMetaData. Additionally a DAO can be used which decides the deployment of new setuTokens and bridges.

Status

Resolved

Comment: The unifarm team said that the owner will be gnosis multisig contract

B.4 ChildToken cannot be upgraded by owner

Description

Although the childToken is an upgradeableToken, it can never be upgraded by the owner of the Registry Contract. It can only be upgraded
by the proxyAdmin. The Registry contract is the owner of the proxyAdmin, not the owner of the Registry contract.

```
function deployChildToken(
    string calldata _name,
    string calldata _imageUrl,
    string calldata _ticker,
    uint8 _decimals
) public onlyOwner {
    require(bridgeTokenMetadata[_ticker].tokenAddress == address(0), "TOKEN_ALREADY_EXISTS");
    TokenUpgradeable newChildToken = new TokenUpgradeable();
    // newChildToken.initialize( name,  ticker,  decimals,  msgSender());

    // deploy ProxyAdmin contract
    ProxyAdmin proxyAdmin = new ProxyAdmin();

    bytes memory data = abi.encodeWithSignature("initialize(string,string,uint8,address)",
                        _name, _ticker, _decimals, _msgSender());
    // deploy TransparentUpgradeableProxy contract
    TransparentUpgradeableProxy transparentUpgradeableProxy = new TransparentUpgradeableProxy(address(newChildToken), address(proxyAdmin), data);

    // transfer ownership of token to bridge
    newChildToken = TokenUpgradeable(address(transparentUpgradeableProxy));
    newChildToken.transferOwnership(bridgeUpgradeable);
```

Remediation

It is advised to review the business and operational logic.

Status

Resolved

Comment: The team transferred the ProxyAdmin ownership to msg.sender (Owner)

B.5 Possibility of revert after _deploySetuToken()

Description

If _deploySetuToken is called externally without adding any bridge via addBridge(), the function disableBridge() will revert always for the setuToken's primary token. Also updateFeeConfig() and updateEpochLength() will not work as it will revert due to the first require check.

```
159     function _deploySetuToken(
160         string memory _name,
161         string memory _ticker,
162         uint8 _decimals
163     ) public onlyOwner {
164         TokenUpgradeable setuToken = new TokenUpgradeable();
165         // setuToken.initialize(concatenate("setu", _name), concatenate("setu", _ticker), _decimals, msgSender());
166 
167         // deploy ProxyAdmin contract
168         ProxyAdmin proxyAdmin = new ProxyAdmin();
169 
170         bytes memory data = abi.encodeWithSignature("initialize(string,string,uint8,address)",
171                             concatenate("setu", _name), concatenate("setu", _ticker), _decimals, _msgSender());
```

Remediation

It is advised to add a check for bridge to be added for a primary token before deployment of a setu token to avoid this issue and review business and operational logic.

Status

Resolved

Comment: The team made this an internal function

Low Severity Issues

B.6 Missing zero address check

Description

Missing zero address check for _newBridgeAddress parameter in the updateBridgeAddress() function. This could lead to transfer of ownership of newChildToken to zero address if deployChildToken() function is called. Similarly if addBridge() function is called, _deploySetuToken() function will be called internally, leading to transfer of ownership of setuToken to zero address.

```
64      function updateBridgeAddress(address _newBridgeAddress) external onlyOwner {
65          bridgeUpgradeable = _newBridgeAddress;
66      }
```

Remediation

Add a require check for the same

Status

Resolved

B.7 Missing require check for feeType

Description

According to the docs feeType can only be 0 or 1. But updateFeeConfig() allows to set fee for any fee type.

```
195     function updateFeeConfig(
196         string calldata _tokenTicker,
197         uint8 _feeType,
198         uint256 _feeInBips
199     ) public onlyOwner {
200         require(tokenBridge[_tokenTicker].startBlock > 0, "NO_TOKEN_BRIDGE_EXISTS");
201         if(_feeType == 1) {
202             _feeInBips *= 100;
203         }
204         FeeConfig memory feeConfig = FeeConfig({
205             feeType: _feeType,
206             feeInBips: _feeInBips
207         });
208         tokenBridge[_tokenTicker].fee = feeConfig;
209     }
```

Remediation

Add necessary checks in updateFeeConfig() for the same

Status

Resolved

B.8 Possibility of Out of Gas errors

Description

It is possible that if the epochLength is set as too small, epochs will get added faster in the Epoch array of the epochs mapping of the BridgeUpgradeable contract. If this happens, this could to out of gas errors when claimFeeShare() and withdrawAdminTokenFees() because the for loop would be looping over a large array length.

```
318     function claimFeeShare(
319         string calldata _tokenTicker,
320         uint256 _index
321     ) public {
322         _isBridgeActive(_tokenTicker);
323 
324         // adding any passed epochs
325         bridgeUpgradeable._addPassedEpochs(_tokenTicker);
326 
327         address _account = _msgSender();
328         (
329             uint256 depositedAmount,
330             ,
331             uint256 claimedTillEpochIndex,
332             ,
333             ,
334             ,
335 
336         ) = bridgeUpgradeable.liquidityPosition(_tokenTicker, _account, _index);
337         require(depositedAmount > 0, "INVALID_POSITION");
338 
339         uint256 startEpoch = claimedTillEpochIndex + 1;
340         uint256 feeEarned = getUserConfirmedRewards(_tokenTicker, _account, _index);
341         require(feeEarned > 0, "NO_REWARD");
342         require(totalFees[_tokenTicker] >= feeEarned, "INSUFFICIENT_FEES");
343         bridgeUpgradeable.updateRewardClaimedTillIndex(_tokenTicker, _account, _index);
344 
345         uint256 endEpoch = bridgeUpgradeable.getEpochsLength(_tokenTicker);
346         for (uint256 epochIndex = startEpoch; epochIndex <= endEpoch; epochIndex++) {
347             bridgeUpgradeable.deleteHasBoosterMapping(_tokenTicker, _account, _index, epochIndex);
348         }
```

```
376     function getAdminTokenFees(
377         string memory _tokenTicker
378     ) public view onlyOwner returns (uint256) {
379         uint256 epochsLength = bridgeUpgradeable.getEpochsLength(_tokenTicker);
380 
381         uint256 adminFees;
382         for (uint256 epochIndex = adminClaimedTillEpoch[_tokenTicker] + 1; epochIndex <= epochsLength; epochIndex++) {
383             uint256 noOfDepositors = bridgeUtilsUpgradeable.getEpochTotalDepositors(_tokenTicker, epochIndex);
384             if(noOfDepositors == 0) {
385                 continue;
386             }
```

```
116     function addBridge(
117         uint8 _bridgeType,
118         string memory _tokenTicker,
119         uint256 _epochLength,
120         uint8 _feeType,
121         uint256 _feeInBips
122     ) public onlyOwner {
123         require(bridgeTokenMetadata[_tokenTicker].tokenAddress != address(0), "TOKEN_NOT_EXISTS");
124         require(tokenBridge[_tokenTicker].startBlock == 0, "TOKEN_BRIDGE_ALREADY_EXISTS");
125         require(_feeType == 0 || _feeType == 1, "INVALID_FEE_TYPE");
```

```
211     function updateEpochLength(
212         string calldata _tokenTicker,
213         uint256 _newEpochLength
214     ) public onlyOwner {
215         require(tokenBridge[_tokenTicker].startBlock > 0, "NO_TOKEN_BRIDGE_EXISTS");
216         tokenBridge[_tokenTicker].epochLength = _newEpochLength;
217     }
```

Remediation

Add a require check for the same

Status

Partially Resolved

Comment: A limit parameter was added to get the rewards for the admin. The team said that they will be adding a similar feature for users as well to resolve this issue.

B.9 Possibility of User paying more fees

Description

There is a possibility of user paying more fees in case fees is changed(and increased) before the transaction reaches the destination chain.

```
195     function updateFeeConfig(
196         string calldata _tokenTicker,
197         uint8 _feeType,
198         uint256 _feeInBips
199     ) public onlyOwner {
200         require(tokenBridge[_tokenTicker].startBlock > 0, "NO_TOKEN_BRIDGE_EXISTS");
201         if(_feeType == 1) {
202             _feeInBips *= 100;
203         }
204         FeeConfig memory feeConfig = FeeConfig({
205             feeType: _feeType,
206             feeInBips: _feeInBips
207         });
208         tokenBridge[_tokenTicker].fee = feeConfig;
209     }
```

Remediation

It is recommended to allow the user to pay fees which was there at the time of transferIn() only. This can also happen in case of replayTransaction()

Status

Acknowledged

Comment: the unifarm team said that they will inform the users about any such fee update in advance, and pause the bridge for some time while performing this operation

Informational Issues

B.10 Missing events for critical functions

Description

Missing events for critical functions like- addTokenMetadata(), _deploySetuToken(), updateFeeConfig(), updateEpochLength(), disableBridgeToken(), enableBridgeToken() and updateNoOfDepositors()

Remediation

It is advised to add events for the same. This would be a best practice for offchain monitoring.

Status

Acknowledged

C. Contract - BridgeUpgradeable

High Severity Issues

C.1 Reetrancy attack in removeLiquidity()

Description

There is possibility of a profitable reentrancy attack in removeLiquidity(). This is because a low level call is used in transferLPFee() function which allows the caller to call any malicious code from another contract. Using this multiple rewards for same liquidity position can be harvested.

For example, Let's say the 1st liquidity position by an LP is 1000 USDC. After that a user does cross-chain transfer. Now the LP sees that he has some rewards available to claim.

Then he adds a 2nd position of say 5000 USDC tokens. Now when the user does a reentrant call via low level call in transferLpFee, he will be able to get twice the reward for the same liquidity position. This is possible because although the index of the first liquidity position will be used for the reentrant call, 2000 or more USDC tokens will need to be withdrawn for claiming more rewards which will be taken from the LP's second liquidity position directly. The rest of the liquidity he will still be able to claim via transferOut.

```
403     function removeLiquidity(
404         uint256 _index,
405         string calldata _tokenTicker
406     ) public isBridgeActive(_tokenTicker) {
407         // adding any passed epochs
408         addPassedEpochs(_tokenTicker);
409 
410         LiquidityPosition memory position = liquidityPosition[_tokenTicker][_msgSender()][_index];
411         require(position.depositedAmount > 0, "INVALID_POSITION");
412 
413         TokenUpgradeable setuToken = getToken(concatenate("setu", _tokenTicker));
414         setuToken.unlockToken(_msgSender(), position.depositedAmount);
415         // setuToken.transfer(_msgSender(), position.depositedAmount);
416 
417         // // Calculate fee share
418         // uint256 feeShare = feePoolUpgradeable.getUserConfirmedRewards(_tokenTicker, _msgSender(), _inc
419 
420         // // Transfer Fees to LP user
421         // // fees token - same as the deposited token OR chain native token
422         // TokenUpgradeable token = getToken( tokenTicker);
423         // token.transferFrom(address(this), _msgSender(), feeShare);
424 
425         feePoolUpgradeable.transferLpFee(_tokenTicker, _msgSender(), _index);
426 
427         // Withdraw liquidity
428         withdrawLiquidity(_index, _tokenTicker);
429 
430         emit LiquidityRemoved(_index, _msgSender(), _tokenTicker);
431     }
```

```
68      function transferLpFee(
69          string memory _tokenTicker,
70          address _account,
71          uint256 _index
72      ) public {
73          require(_msgSender() == address(bridgeUpgradeable), "ONLY_BRIDGE");
74          // Calculate fee share
75          uint256 feeShare = getUserConfirmedRewards(_tokenTicker, _account, _index);
76          require(totalFees[_tokenTicker] >= feeShare, "INSUFFICIENT_FEES");
77
78          uint8 feeType;
79          (feeType, ) = bridgeUtilsUpgradeable.getFeeTypeAndFeeInBips(_tokenTicker);
80          totalFees[_tokenTicker] -= feeShare;
81
82          // Transfer Fees to LP user
83          if(feeType == 0) {
84              // fee in native chain token
85              (bool success, ) = _account.call{value: feeShare}("");
86              require(success, "TRANSFER_FAILED");
87          }
88          else if(feeType == 1) {
89              TokenUpgradeable token = bridgeUpgradeable.getToken(_tokenTicker);
90              token.transfer(_account, feeShare);
91          }
92
93      }
```

Remediation

It is advised to use transfer or send instead of low level call in transferLPFee() function and follow a checks effects interactions pattern in removeLiquidity() function.

Status

Partially Resolved

Medium Severity Issues

C.2 Centralization of safeWithdrawLiquidity()

Description

safeWithdrawLiquidity() can be used by a malicious owner to withdraw all the liquidity of any token.

```
678     function safeWithdrawLiquidity(
679         string calldata _tokenTicker,
680         uint256 _noOfTokens
681     ) external onlyOwner {
682         // require(_noOfTokens <= totalLiquidity[_tokenTicker], "AMOUNT_OVERFLOW");
683         // totalLiquidity[_tokenTicker] -= _noOfTokens;
684 
685         TokenUpgradeable token = getToken(_tokenTicker);
686         token.transfer(owner(), _noOfTokens);
687     }
```

Remediation

It is advised to utilize a multisg wallet of at least ¾ configuration to make calling of this function more decentralized and safer.

Status

Resolved

C.3 removeBridge() can lead to Denial of Service

Description

If an existing bridge is removed using removeBridge() which contains liquidity provided by LPs, they will not be able to claim back their liquidity. This would also result in LPs unable to claim any fees accumulated till that point as well.

```
292   function removeBridge(string memory _tokenTicker) external onlyOwner {
293       delete tokenBridge[_tokenTicker];
294
295       uint256 len = tokenBridges.length;
296       for (uint256 index = 0; index < len; index++) {
297           // string memory token = tokenBridges[index];
298           if(keccak256(abi.encodePacked(tokenBridges[index])) == keccak256(abi.encodePacked(_tokenTicker))) {
299               if(index < len-1) {
300                   tokenBridges[index] = tokenBridges[len-1];
301               }
302               tokenBridges.pop();
303               break;
304           }
305       }
306   }
307
308   receive() external payable {}
309 }
```

Remediation

It is advised to allow the removeBridge() to be called for a particular tokenTicker only when fee rewards have been distributed for the bridge, and only when the existing LPs have been given back their liquidity

Status

Resolved

C.4 Centralization of getBackTokens()

Description

getBackTokens() can be exploited by malicious admin to drain the liquidity from any liquidity pool.

```
737     function getBackTokens(address tokenAddress) external onlyOwner {
738         IERC20 token = IERC20(tokenAddress);
739         token.transfer(msg.sender, token.balanceOf(address(this)));
740     }
741
742     function getBackNativeTokens() external onlyOwner {
743         (bool success, ) = msg.sender.call{value: address(this).balance}("");
744         require(success, "TRANSFER_FAILED");
745     }
```

Remediation

It is advised to remove the function if not required and review business logic.

Status

Partially Resolved

C.5 Loss of Boosters

Description

If a user buys booster packs using buyBoosterPacks() and then add liquidity using addLiquidity(), then the user will lose his currently bought boosters due to overwriting of boosterEndEpochIndex for that particular liquidity position.

For example, let's say user buys 1000 booster packs for token USDC, with _index parameter specified as 1 (for which he has not added any liquidity yet and let's say currentEpochIndex is 50). This will set boosterEndEpochIndex to 1049. Now at epoch say 60, and the user adds liquidity using addLiquidity() with _noOfBoosters as zero, the boosterEndEpochIndex will get overwritten to 60 resulting in loss of boosters bought.

```
737     function getBackTokens(address tokenAddress) external onlyOwner {
738         IERC20 token = IERC20(tokenAddress);
739         token.transfer(msg.sender, token.balanceOf(address(this)));
740     }
741
742     function getBackNativeTokens() external onlyOwner {
743         (bool success, ) = msg.sender.call{value: address(this).balance}("");
744         require(success, "TRANSFER_FAILED");
745     }
```

Remediation

Consider reviewing business logic, It is advised to add specific checks to ensure that this does not happen.

Status

Resolved

Low Severity Issues

C.6 Inheritance by utils contract not required

Description

BridgeUpgradeableUtils contract inherits BridgeStorage contract which is not required to be used anywhere.

```
12  contract BridgeUtilsUpgradeable is Initializable, OwnableUpgradeable, RegistryStorage, BridgeStorage {
13
14      TokenBridgeRegistryUpgradeable public tokenBridgeRegistryUpgradeable;
15
16      BridgeUpgradeable public bridgeUpgradeable;
17
18      FeePoolUpgradeable public feePoolUpgradeable;
```

Remediation

It is advised to remove the redundant inherited contract.

Status

Acknowledged

Comment: The team said that as the size of registry and bridge contract is quite large, they need an extra contract for utility or read only functions

C.7 Reentrancy in transferOut()

Description

A reentrant attack is possible via transferOut() function because of the low level call used in calculateBridgingFees() function. The low level call allows another account to call malicious code although in this case it would just be a griefing attack as a simple reentrant call would only result in the caller paying more fees.

But there is still the possibility of a cross-contract reentrancy attack if the contracts in question used shared variables which can not even be mitigated by using non-Reentrant modifier from Openzeppelin.

Note that the function crossChainTransferIn() will also be vulnerable to the same attack as transferOut is being called internally there.

```
597     function transferOut(
598         // address _userAddress,
599         uint256 _noOfTokens,
600         string calldata _tokenTicker
601     ) public payable isBridgeActive(_tokenTicker) {
602         // adding any passed epochs
603         addPassedEpochs(_tokenTicker);
604 
605         TokenUpgradeable setuToken = getToken(concatenate("setu", _tokenTicker));
606         TokenUpgradeable token = getToken(_tokenTicker);
607 
608         uint8 bridgeType = bridgeUtilsUpgradeable.getBridgeType(_tokenTicker);
609 
610         uint256 currentLiquidity = totalLiquidity[_tokenTicker];
611         uint256 noOfTokens = _noOfTokens;
612         // pool has less liquidity
613         if(currentLiquidity < _noOfTokens) {
614             noOfTokens = currentLiquidity;
615         }
616 
617         // 0 - liquidity bridge, 1 - child + liquidity bridge
618         if(bridgeType == 0) {
619             uint256 feesDeducted = calculateBridgingFees(_tokenTicker, noOfTokens);
620             setuToken.burnTokens(_msgSender(), noOfTokens);
621             totalLiquidity[_tokenTicker] -= noOfTokens;
622             token.transfer(_msgSender(), noOfTokens - feesDeducted);
623             if(feesDeducted > 0) {
624                 token.transfer(address(feePoolUpgradeable), feesDeducted);
625             }
626         }
```

```
651     function calculateBridgingFees(
652         string calldata _tokenTicker,
653         uint256 _noOfTokens
654     ) internal returns (uint256) {
655         uint8 feeType;
656         uint256 feeInBips;
657         uint256 fees;
658         (feeType, feeInBips) = bridgeUtilsUpgradeable.getFeeTypeAndFeeInBips(_tokenTicker);
659 
660         // fee in native chain token
661         if(feeType == 0) {
662             require(msg.value >= feeInBips, "INSUFFICIENT_FEES");
663             feePoolUpgradeable.updateTotalFees(_tokenTicker, feeInBips, true);
664             (bool success, ) = address(feePoolUpgradeable).call{value: feeInBips}("");
665             require(success, "POOL_TRANFER_FAILED");
666 
667             (success, ) = _msgSender().call{value: msg.value - feeInBips}("");
668             require(success, "SENT_BACK_FAILED");
669         }
```

Remediation

Consider reviewing business logic, It is advised to add specific checks to ensure that this does not happen.

Status

Resolved

C.8 Possibility of mismatch of contract addresses

Description

In the BridgeUtilsUpgradeable contract, there is a possibility of mismatch between the feePoolUpgradeable and the most current FeePoolUpgradeable contract in usage as well as bridgeUpgradeable and the most current BridgeUpgradeable in usage. This is because the owner of the contract can forget to change the addresses of the bridge and the feepool contract after they have been upgraded. A malicious admin can also take an advantage of this to hide the actual contract with some malicious code.

```
12 contract BridgeUtilsUpgradeable is Initializable, OwnableUpgradeable, RegistryStorage, BridgeStorage {
13
14     TokenBridgeRegistryUpgradeable public tokenBridgeRegistryUpgradeable;
15
16     BridgeUpgradeable public bridgeUpgradeable;
17
18     FeePoolUpgradeable public feePoolUpgradeable;
```

Remediation

It is advised to remove the redundant inherited contract.

Status

Acknowledged

C.9 Redundant code

Description

The if statement checks whether the mapping is zero and then deletes the mapping. But this is not needed because delete does the same thing as setting the mapping to zero value.

```
588        if(transferMapping[key] == 0) {
589            delete transferMapping[key];
590        }
591
592        transferOut(_noOfTokens, _tokenTicker);
```

Remediation

It is advised to remove the redundant code

Status

Resolved

C.10 _gap variable not required

Description

__gap variable is not required in BridgeUpgradeable contract (declared at the end of the contract) because __gap variables are generally used in Base contracts and not in Child contracts. It may lead to unintended issues or mistakes.

```
746
747        receive() external payable {}
748
749        uint256[100] private __gap;
750    }
```

Remediation

It is advised to remove the __gap variable and review business and operational logic

Status

Resolved

C.11 updateBoosterConfig() lacks sanity checks

Description

The updateBoosterConfig() function lacks sanity checks for its parameters such as _perBoosterPrice and _adminAccount and thus can be accidentally set to incorrect values.

```
199     function updateBoosterConfig(
200         address _adminAccount,
201         address _boosterToken,
202         uint256 _perBoosterPrice,
203         string calldata _imageUrl
204     ) external onlyOwner {
205         boosterConfig = BoosterConfig({
206             tokenAddress: _boosterToken,
207             price: _perBoosterPrice,
208             imageUrl: _imageUrl,
209             adminAccount: _adminAccount
210         });
211     }
```

Remediation

It is advised to add sanity value checks for updateBoosterConfig() function parameters.

Status

Resolved

Informational Issues

C.12 Missing event for safeWithdrawLiquidity()

Description

safeWithdrawLiquidity is a crucial function which does not emit any event.

```
678     function safeWithdrawLiquidity(
679         string calldata _tokenTicker,
680         uint256 _noOfTokens
681     ) external onlyOwner {
682         // require(_noOfTokens <= totalLiquidity[_tokenTicker], "AMOUNT_OVERFLOW");
683         // totalLiquidity[_tokenTicker] -= _noOfTokens;
684 
685         TokenUpgradeable token = getToken(_tokenTicker);
686         token.transfer(owner(), _noOfTokens);
687     }
```

Remediation

It is advised to add event for the same which would be a best practice for offchain monitoring.

Status

Acknowledged

D. Contract - FeePoolUpgradeable

High Severity Issues

No issues found

Medium Severity Issues

D.1 Centralization of update functionality of tokenBridgeRegistryUpgradeable and bridgeUpgradeable address

Description

tokenBridgeRegistryUpgradeable and bridgeUpgradeable can be updated anytime by the owner using updateTokenBridgeRegistryAddress() and updateBridgeUpgradeableAddress().

If an LP has fees accrued say 100 USD in the contract, but the BridgeUpgradeable address is updated using updateBridgeUpgradeableAddress() function, then the LP will never be able to claim this fees- leading to a Denial of Service.

In the BridgeUtilsUpgradeable contract, there is a possibility of mismatch between the feePoolUpgradeable and the most current FeePoolUpgradeable contract in usage as well as bridgeUpgradeable and the most current BridgeUpgradeable in usage, which is same as the issue stated in BridgeUpgradeable contracts section above in the report.

```
56      function updateTokenBridgeRegistryAddress(TokenBridgeRegistryUpgradeable _newInstance) external onlyOwner {
57          tokenBridgeRegistryUpgradeable = _newInstance;
58      }
59
60      function updateBridgeUpgradeableAddress(BridgeUpgradeable _newInstance) external onlyOwner {
61          bridgeUpgradeable = _newInstance;
62      }
63
64      function updateBridgeUtilsUpgradeableAddress(BridgeUtilsUpgradeable _newInstance) external onlyOwner {
65          bridgeUtilsUpgradeable = _newInstance;
```

Remediation

It is advised to automatically transfer the fees to the previous LPs when updating the address of the contract. Or add additional features for the old LPs to claim their fees previously accrued. Also steps should be taken to ensure that there is no mismatch of contract addresses of FeePool and Bridge addresses set in BridgeUtilsUpgradeable otherwise it is advised to remove the utils contract and to make the contract interactions simpler.

Status

Acknowledged

Comment: The team said that changing back the address to the correct one will fix this.

Low Severity Issues

D.2 getUserConfirmedRewards() shows incorrect rewards

Description

getUserConfirmedRewards() shows zero rewards even when there are non-zero rewards claimable by liquidity providers. This is because the fee calculation and distribution depends on addPassedEpochs() that gets updated only when claimFeeShare, addLiquidity, etc. is called. This means that Liquidity Providers don't have a foolproof way to check what their current rewards are using getUserConfirmedRewards() or even getUserUnconfirmedRewards() unless someone calls claimFeeShare or other state changing functions which call addPassedEpochs() internally.

```
267     function getUserConfirmedRewards(
268         string memory _tokenTicker,
269         address _account,
270         uint256 _index
271     ) public view returns (uint256) {
272         (
273             uint256 depositedAmount,
274             uint256 blockNo,
275             uint256 claimedTillEpochIndex,
276             uint256 epochStartIndex,
277             uint256 epochStartBlock,
278             ,
279 
280         ) = bridgeUpgradeable.liquidityPosition(_tokenTicker, _account, _index);
281         require(depositedAmount > 0, "INVALID_POSITION");
282 
283         uint256 epochsLength = bridgeUpgradeable.getEpochsLength(_tokenTicker);
284 
285         uint256 feeEarned;
286 
287         // user still in starting epoch
288         if(epochsLength == epochStartIndex - 1) {
289             return feeEarned;
290         }
```

Remediation

It is advised to add either mock addPassedEpochs() calculation in view function so that LPs can check if they have any rewards or provide a safe method to addPassedEpochs after certain intervals of time.

Status

Acknowledged

Comment: The team said that they were okay with this issue currently.

Informational Issues

D.3 Unused internal function

Description

getLPsFee() is an internal function that is not getting called anywhere.

```
    function getLPsFee(
        uint256 epochTotalFees,
        string memory tokenTicker,
        uint256 epochIndex
    ) internal view returns (uint256) {
        uint256 totalBoostedUsers = bridgeUpgradeable.totalBoostedUsers(tokenTicker, epochIndex);
        uint256 noOfDepositors;
        uint256 epochsLength = bridgeUpgradeable.getEpochsLength(tokenTicker);
        // calculation for the first epoch or the current ongoing epoch
        if((epochIndex == 1 && epochsLength == 0) || epochIndex == epochsLength + 1) {
            noOfDepositors = bridgeUtilsUpgradeable.getNoOfDepositors(tokenTicker);
        } else {
            noOfDepositors = bridgeUtilsUpgradeable.getEpochTotalDepositors(tokenTicker, epochIndex);
        }

        uint256 perUserFee = epochTotalFees / (2 * noOfDepositors);

        uint256 normalFees = (noOfDepositors - totalBoostedUsers) * perUserFee;
        uint256 extraBoostedFees = totalBoostedUsers * perUserFee * 3 / 2;
        return (extraBoostedFees + normalFees);
    }
```

Remediation

It is advised to remove this function and review business logic if necessary.

Status

Resolved

E. Contract - Common Issues

High Severity Issues

No issues found

Medium Severity Issues

E.1 Insufficient test coverage

Description

There is insufficient test coverage provided for the codebase. For some situations it is possible that router protocol should revert or do expected things which must be checked while in testing on mainnet.

For example, It is possible that routerSend() returns true but the transaction fails to go on the destination chain. If it passes and the dest chain tx could not be executed due to low gas price and limit, then replay transaction needs to be done.

Remediation

We recommend to have at least 80 percent of test coverage for the codebase. Also it is recommended to run these tests on mainnet as it cannot be checked locally

Status

Acknowledged

Low Severity Issues

E.2 Renounce ownership

Description

The renounceOwnership function can be called accidentally by the admin leading to immediate renouncement of ownership to zero address after which
any onlyOwner functions will not be callable which can be risky.

Remediation

It is advised that the Owner cannot call renounceOwnership without first transferring ownership to a different address. Additionally, if a multi-signature wallet is utilized, executing the renounceOwnership method for two or more users should be confirmed. Alternatively, the Renounce Ownership functionality can be disabled by overriding it.

Status

Resolved

Comment: The team said that the owner will be gnosis multisig contract

E.3 Transfer ownership

Description

The transferOwnership() function in contract allows the current admin
to transfer his privileges to another address. However, inside transferOwnership() , the newOwner is directly stored into the storage owner, after validating the newOwner is a non-zero address, and immediately overwrites the current owner. This can lead to cases where the admin has transferred ownership to an incorrect address and wants to revoke the transfer of ownership or in the cases where the current admin comes to know that the new admin has lost access to his account.

Remediation

It is advised to make ownership transfer a two-step process.

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol

Status

Resolved

E.4 Naming conventions not followed thoroughly

Description

Solidity naming conventions are not followed thoroughly. For example: Internal function should be named _addPassedEpochs() and external function should be named addPassedEpochs(). Also as _deploySetuToken() is a public function, it should be named deploySetuToken() instead.

Remediation

It is advised to follow solidity naming conventions thoroughly.

Status

Resolved

E.5 Possibility of reentrancy

Description

The functions addLiquidity(), crossChainTransferIn(), crossChainTransferOut(), transferIn() can be vulnerable to reentrancy attack if setutoken is malicious.

Remediation

It is advised to follow checks and effects interactions patterns and avoid usage of any low level calls.

Status

Resolved

Informational Issues

E.6 Possibility of exploit of createCrossChainTransferMapping()

Description

The function createCrossChainTransferMapping() of the BridgeUpgradeable contract which is critical for cross chain transfers, relies highly Router protocol. There is a possibility that the _transferIndex parameter is set acccordingly to pass the require check every time in createCrossChainTransferMapping(). Also there can be a possibility of transactions getting manipulated or denied of getting executed crosschain if Router protocol is centralized or gets compromised.

Remediation

It is advised to understand the risks of it and improve the safety measures and checks to ensure that the bridge remains safe in the long run

Status

Acknowledged

F. General Recommendations Summary

1) Floating pragma usage should be avoided. Usage a fixed compiler version instead.
2) Low level call should be replaced with transfer or send
3) Testing needs to be performed with good coverage to check if protocol used for bridging is reverting and giving expected output on certain conditions
4) Use multisig wallets
5) Perform rigorous testing on Mainnet
6) Use analytics and real time monitoring tools to mitigate any suspicious activities or attacks as soon as possible

G. Contract - Automated Tests

High Severity Issues

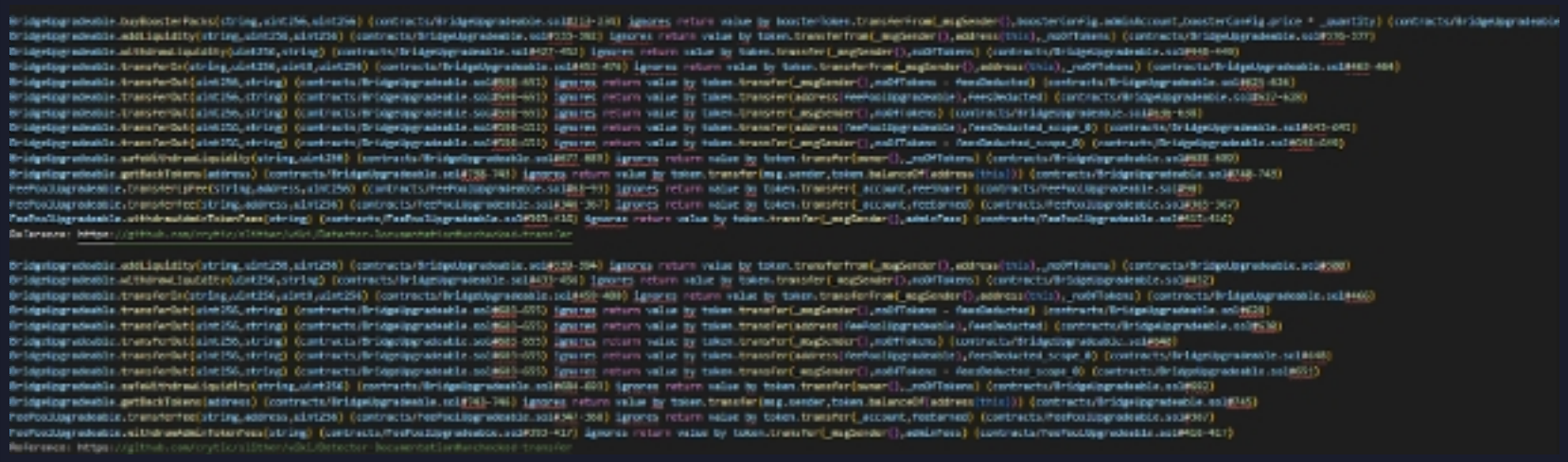
No issues found

Medium Severity Issues

G.1 Unchecked return value

Description

Token transfers in functions such addLiquidity(), withdrawLiquidity(), transferLPFee() and transferOut() utilize the transfer and transferFrom function of ERC20. But these transfers return a boolean value which is not checked. Thus in cases of tokens which do not revert on failed token transfer, it could lead to successful execution of these functions even though the transfer fails.



Remediation

It is advised to add a require check for boolean value returned by token transfers specified in the image above.

Status

Acknowledged

Low Severity Issues

No issues found

Informational Issues

No issues found

Closing Summary

In this report, we have considered the security of the Unifarm. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture. In the End, Unifarm Team Resolved most of Issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Unifarm Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the UnifarmTeam put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

600+
Audits Completed

$15B
Secured

600K
Lines of Code Audited

Follow Our Journey

Audit Report
November, 2022

For

UniFarm

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com