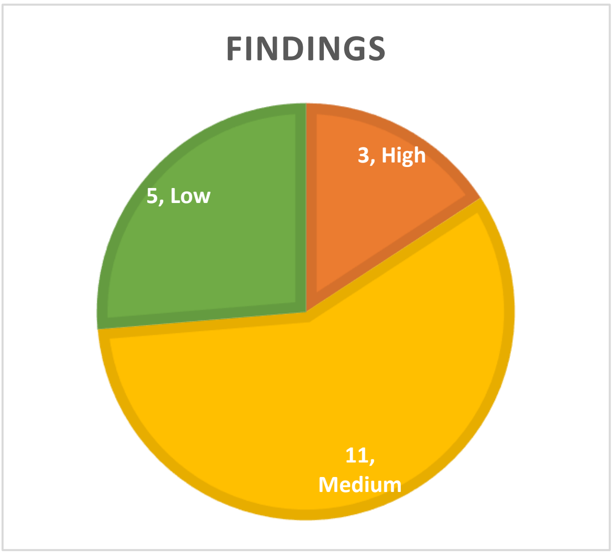
Trust Security

Smart Contract Audit

Mozaic Archimedes

23/05/2023

Executive summary

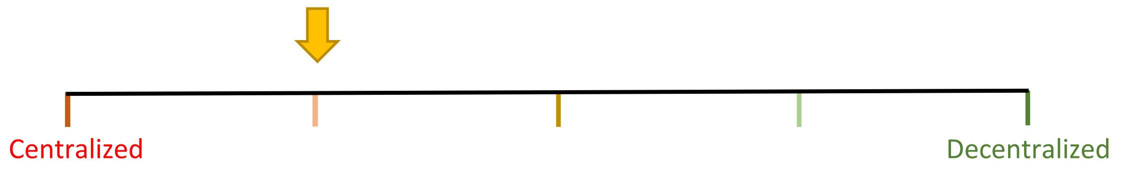


Category	Yield Aggregator
Audited file count	9
Lines of Code	1460
Auditor	Trust
Time period	10/05-23/05

Findings

Severity	Total	Open	Fixed	New issues	Acknowledged
High	3	-	3	-	-
Medium	11	3	8	1	-
Low	5	1	3	1	1

Centralization score



Signature

Digitally signed by Trust
Date: 2023-06-05 12:36:48

Document properties

Versioning

Version	Date	Description
0.1	23/05/23	Client report
0.2	05/06/23	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- contracts/Vault.sol
- contracts/MozBridge.sol
- contracts/StargatePlugin.sol
- contracts/Controller.sol
- contracts/MozaicLP.sol
- contracts/MozStaking.sol
- contracts/MozToken.sol
- contracts/MultiSig.sol
- contracts/XMozToken.sol

Scope notes

Between the initial audit and the mitigation review, Mozaic has refactored the cross-chain messaging logic, to support delivery re-tries. **All related changes** in Controller.sol, MozBridge.sol and Vault.sol are **not in scope** for the audit.

Repository details

- **Repository URL:** https://github.com/Mozaic-fi/mozaic-contract-poc
- **Commit hash #1:** 9f2b2c3ad6f33b100efe550ac6ad43a3532a885c
- **Commit hash #2:** 31f73431cde8f018b0a089b352ffd2618d3d0058
- **Mitigation commit hash #1:** 4a695988be0e69ae5d74859583f537ae2dc1c022
- **Mitigation commit hash #2:** 995713a0094eee2f7d5cde6617314284203ac0de

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is a leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	**Rating**	**Comments**
Code complexity	**Good**	Project has managed to keep the code as simple as possible, reducing attack risks
Documentation	**Good**	Project is mostly well documented.
Best practices	**Good**	Project is aware of industry standards and mostly follows them well.
Centralization risks	**Moderate**	In its current iteration, Mozaic holds significant power in management of user funds.

Findings

High severity findings

TRST-H-1 An attacker can drain Mozaic Vaults by manipulating the LP price

- **Category:** Economic attacks
- **Source:** Vault.sol
- **Status:** Fixed + *require attention*

Description

The controller is tasked with synchronizing LP token price across all chains. It implements a lifecycle. An admin initiates the snapshot phase, where Controller requests all Vaults to report the total stable ($) value and LP token supply. Once all reports are in, admin calls the settle function which dispatches the aggregated value and supply to all vaults. At this point, vaults process all deposits and withdrawals requested up to the last snapshot, using the universal value/supply ratio.

The described pipeline falls victim to an economic attack, stemming from the fact that LP tokens are LayerZero OFT tokens which can be bridged. An attacker can use this property to bypass counting of their LP tokens across all chains. When the controller would receive a report with correct stable value and artificially low LP supply, it would cause queued LP withdrawals to receive incorrectly high dollar value.

To make vaults miscalculate, attacker can wait for Controller to initiate snapshotting. At that moment, they can start bridging a large amount of tokens. They may specify custom LayerZero adapter params to pay a miniscule gas fee, which will guarantee that the bridge-in transaction will fail due to out-of-gas. At this point, they simply wait until all chains have been snapshotted, and then finish bridging-in with a valid gas amount. Finally, Controller will order vaults to settle, at which point the attacker converts their LP tokens at an artificially high price.

Another clever way to exploit this flaw is to count LP tokens multiple times, by quickly transporting them to additional chains just before those chains are snapshotted. This way, the LP tokens would be diluted and the attacker can get a disproportionate amount of LP tokens for their stables.

Recommended mitigation

The easy but limiting solution is to reduce complexity and disable LP token bridging across networks. The other option is to increase complexity and track incoming/outgoing bridge requests in the LP token contract. When snapshotting, cross reference the requested snapshot time with the bridging history.

Team response

Fixed.

Mitigation review

Mozaic's response is to disable LP token bridging altogether. As this is a configuration-level fix, users are encouraged to confirm the tokens are not linked via bridges at runtime.

TRST-H-2 Attacker can freeze deposits and withdrawals indefinitely by submitting a bad withdrawal

- **Category:** Logical flaws
- **Source:** Vault.sol
- **Status:** Fixed

Description

Users request to queue a withdrawal using the function below in Vault.

```
function addWithdrawRequest(uint256 _amountMLP, address _token)
external {
    require(isAcceptingToken(_token), "ERROR: Invalid token");
    require(_amountMLP != 0, "ERROR: Invalid amount");

    address _withdrawer = msg.sender;
    // Get the pending buffer and staged buffer.
    RequestBuffer storage _pendingBuffer = _requests(false);
    RequestBuffer storage _stagedBuffer = _requests(true);
    // Check if the withdrawer have enough balance to withdraw.
    uint256 _bookedAmountMLP =
_stagedBuffer.withdrawAmountPerUser[_withdrawer] +
_pendingBuffer.withdrawAmountPerUser[_withdrawer];
    require(_bookedAmountMLP + _amountMLP <=
MozaicLP(mozLP).balanceOf(_withdrawer), "Withdraw amount > amount
MLP");
    …
    emit WithdrawRequestAdded(_withdrawer, _token, chainId,
_amountMLP);
}
```

Notice that the function only validates that the user has a sufficient LP token balance to withdraw at the moment of execution. After it is queued up, a user can move their tokens to another wallet. Later in *_settleRequests(),* the Vault will attempt to burn user's tokens:

```
// Burn moazic LP token.
MozaicLP(mozLP).burn(request.user, _mlpToBurn);
```

This would revert and block any other settlements from occurring. Therefore, users can block the entire settlement process by requesting a tiny withdrawal amount in every epoch and moving funds to another wallet.

Recommended mitigation

Vault should take custody of user's LP tokens when they request withdrawals. If the entire withdrawal cannot be satisfied, it can refund some tokens back to the user.

Team response

Fixed.

Mitigation review

The Vault now holds custody of withdrawn LP tokens. If it is not able to transfer the desired amount of stablecoins, it will transfer the remaining LP tokens back to the user.

TRST-H-3 All LayerZero requests will fail, making the contracts are unfunctional

- **Category:** Gas-related issues
- **Source:** Controller.sol,Vault.sol,StargatePlugin.sol
- **Status:** Fixed

Description

When sending messages using the LayerZero architecture, native tokens must be supplied to cover the cost of delivering the message at the receiving chain. However, none of the Mozaic contracts account for it. The controller calls the bridge's *requestSnapshot()*, *requestSettle()*, *requestExecute()* without passing value. Vault calls *reportSnapshot(), reportSettle()* similarly. StargatePlugin calls the StargateRouter's *swap()* which also requires value. As a result, the contracts are completely unfunctional.

Recommended mitigation

Pass value in each of the functions above. Perform more meticulous testing with LayerZero endpoints. Contracts should support receiving base tokens with the receive() fallback, to pay for fees.

Team response

Fixed.

Mitigation review

The Controller and Vault now pass appropriate value in native tokens for messaging. The contracts can be topped-up with the *receive()* method.

Medium severity findings

TRST-M-1 Removal of Multisig members will corrupt data structures

- **Category:** Off-by-one errors
- **Source:** Multisig.sol
- **Status:** Fixed

Description

The Mozaic Multisig (the senate) can remove council members using the **TYPE_DEL_OWNER** operation:

```
if(proposals[_proposalId].actionType == TYPE_DEL_OWNER) {
    (address _owner) = abi.decode(proposals[_proposalId].payload,
(address));
    require(contains(_owner) != 0, "Invalid owner address");
    uint index = contains(_owner);
    for (uint256 i = index; i < councilMembers.length - 1; i++) {
        councilMembers[i] = councilMembers[i + 1];
    }
    councilMembers.pop();
    proposals[_proposalId].executed = true;
    isCouncil[_owner] = false;
}
```

The code finds the owner's index in the councilMembers array, copies all subsequent members downwards, and deletes the last element. Finally, it deletes the **isCouncil[_owner]** entry.

The issue is actually in the *contains()* function.

```
function contains(address _owner) public view returns (uint) {
    for (uint i = 1; i <= councilMembers.length; i++) {
        if (councilMembers[i - 1] == _owner) {
            return i;
        }
    }
    return 0;
}
```

The function returns the index *following* the owner's index. Therefore, the intended **owner** is not deleted from **councilMembers**, instead the one after it is. The *submitProposal()* and *confirmTransaction()* privileged functions will not be affected by the bug, as they filter by **isCouncil**. However, the corruption of **councilMembers** will make deleting the member following the currently deleted owner fail, as deletion relies on finding the member in **councilMembers**.

Recommended mitigation

Fix the *contains()* function to return the correct index of **_owner**.

Team response

Fixed.

Mitigation review

Index is calculated correctly.

TRST-M-2 Multisig could become permanently locked

- **Category:** Validation issues
- **Source:** Multisig.sol
- **Status:** Open (Partial fix)

Description

As described, the senate can remove council members. It can also adjust the threshold for quorum using the **TYPE_ADJ_THRESHOLD** proposal type. Both remove and adjust operations do not perform an important security validation, that the new council member count and threshold number allow future proposal to pass.

Recommended mitigation

Verify that **councilMembers.length >= threshold**, after execution of the proposal.

Team response

Fixed.

Mitigation review

The TYPE_ADJ_THRESHOLD proposal now checks the new threshold is safe. However it is not checked during owner removal.

TRST-M-3 Attacker could abuse victim's vote to pass their own proposal

- **Category:** Reorg attacks
- **Source:** Multisig.sol
- **Status:** Fixed

Description

Proposals are created using *submitProposal()*:

```
function submitProposal(uint8 _actionType, bytes memory _payload)
public onlyCouncil {
    uint256 proposalId = proposalCount;
    proposals[proposalId] = Proposal(msg.sender,_actionType,
_payload, 0, false);
    proposalCount += 1;
    emit ProposalSubmitted(proposalId, msg.sender);
}
```

After submission, council members approve them by calling *confirmTransaction()*:

```
function confirmTransaction(uint256 _proposalId) public onlyCouncil
notConfirmed(_proposalId) {
    confirmations[_proposalId][msg.sender] = true;
    proposals[_proposalId].confirmation += 1;
    emit Confirmation(_proposalId, msg.sender);
}
```

Notably, the **_proposalId** passed to *confirmTransaction()* is simply the **proposalCount** at time of submission. This design allows the following scenario to occur:

1. User A submits proposal P1
2. User B is interested in the proposal and confirms it
3. Attacker submits proposal P2
4. A blockchain re-org occurs. Submission of P1 is dropped in place of P2.
5. User B's confirmation is applied on top of the re-orged blockchain. Attacker gets their vote.

We've seen very large re-orgs in top blockchains such as Polygon, so this threat remains a possibility to be aware of.

Recommended mitigation

Calculate **proposalId** as a hash of the proposal properties. This way, votes cannot be misdirected.

Team response

Fixed.

Mitigation review

The suggestion mitigation has been applied correctly. It is woth noting that the new **proposalIds** array will keep growing throughout the governance lifetime. At some point, it may be too large to fetch using *getProposalIds()*.

TRST-M-4 Users can lose their entire xMoz balance when specifying too short a duration for redemption

- **Category:** Leak of value issues
- **Source:** MozStaking.sol
- **Status:** Fixed

Description

Users can convert their XMoz to Moz through MozStaking, using *redeem()*.

```
function redeem(uint256 xMozAmount, uint256 duration) external  {
    require(xMozAmount > 0, "redeem: xMozAmount cannot be zero");
    xMozToken.transferFrom(msg.sender, address(this), xMozAmount);
    uint256 redeemingAmount = xMozBalances[msg.sender];
    // get corresponding MOZ amount
    uint256 mozAmount = getMozByVestingDuration(xMozAmount, 
duration);
    if (mozAmount > 0) {
        emit Redeem(msg.sender, xMozAmount, mozAmount, duration);
        // add to total
        xMozBalances[msg.sender] = redeemingAmount + xMozAmount;
        // add redeeming entry
        userRedeems[msg.sender].push(RedeemInfo(mozAmount, 
xMozAmount, _currentBlockTimestamp() + duration));
    }
}
```

If the specified duration is shorter than the **minRedeemDuration** specified in the staking contract, **mozAmount** will end up being zero. In such scenarios, redeem will consume user's **xMozAmount** without preparing any redemption at all. The contract should not expose an interface that so easily can lead to loss of funds.

Recommended mitigation

If **duration** is less than **minRedeemDuration**, revert the transaction.

Team response

Fixed.

Mitigation review

The staking contract now verifies that the staking duration is safe to use.

TRST-M-5 MozStaking does not reserve Moz tokens for redemptions, leading to unfulfillable redemptions.

- **Category:** Logical flaws
- **Source:** MozStaking.sol
- **Status:** Fixed

Description

When users call *redeem()* in MozStaking, they are scheduling a future redemption for a specific Moz amount. However, that amount is not set aside for them. Other users can "cut in line", request a redemption for a lower duration and empty the Moz bank. The original user would have to cancel redemption or wait until new users stake their Moz.

The assumption that **Moz supply > XMoz supply** in the staking contract does not hold, as there is an initial XMoz supply minted in XMozToken.

Recommended mitigation

Another state variable should be introduced to account for the reserved Moz amount. MozStaking should not allow new redemptions if there is currently insufficient Moz.

Team response

Fixed.

Mitigation review

The staking contract mints and burns tokens, ensuring it cannot run out of them.

TRST-M-6 MozToken will have a much larger fixed supply than intended.

- **Category:** Initialization flaws
- **Source:** MozToken.sol
- **Status:** Fixed + *require attention*

Description

MozToken is planned to be deployed on all supported chains. Its total supply will be 1B. However, its constructor will mint 1B tokens on *each* deployment.

```
constructor(
   address _layerZeroEndpoint,
   uint8 _sharedDecimals
) OFTV2("Mozaic Token", "MOZ", _sharedDecimals, _layerZeroEndpoint) {
   _mint(msg.sender, 1000000000 * 10 ** _sharedDecimals);
   isAdmin[msg.sender] = true;
}
```

Recommended mitigation

Pass the minted supply as a parameter. Only on the main chain, mint 1B tokens.

Team response

Fixed.

Mitigation review

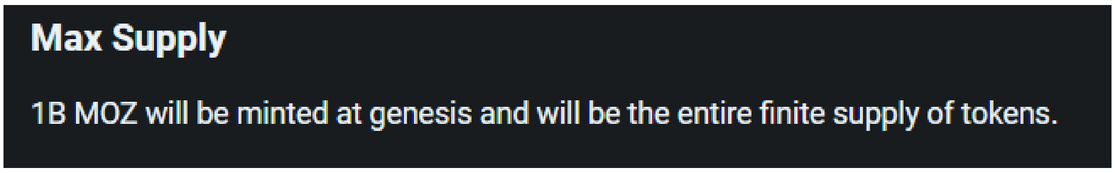
According to Mozaic, Moz and XMoz tokens will be deployed on base chain with the contracts audited. When deploying on additional chains, they will remove the *_mint()* call.

TRST-M-7 MozToken allows owner to mint an arbitrary amount of tokens, although supply is fixed

- **Category:** Economic issues
- **Source:** MozToken.sol
- **Status:** Open (Partial fix)

Description

Supply of MozToken is defined to be fixed at 1B tokens:



Max Supply

1B MOZ will be minted at genesis and will be the entire finite supply of tokens.

However, there is an exposed *mint()* function which allows the owner to mint arbitrary amount of tokens.

```
function mint(address to, uint256 amount) public onlyOwner {
    _mint(to, amount);
}
```

This would typically be in the centralization risks section, however due to the fact that the documentation is potentially misleading users, it must appear in the main report as well.

Recommended mitigation

Remove the *mint()* function.

Team response

Fixed.

Mitigation review

The fix made only the staking contract capable of minting tokens.

```
function mint(uint256 _amount, address _to) external
onlyStakingContract {
    _mint(_to, _amount);
}
```

However, the centralization issue remains as the owner can change the staking contract at once.

```
function setStakingContract(address _mozStaking) external onlyOwner {
    require(_mozStaking != address(0x0), "Invalid address");
    mozStaking = _mozStaking;
}
```

TRST-M-8 The vault cannot operate with popular non-conforming ERC20 tokens due to unsafe transfers

- **Category:** ERC20 compatibility flaws
- **Source:** Vault.sol
- **Status:** Open (Partial fix)

Description

In Vault, the admin can deposit and withdraw tokens using the functions below:

```
///@notice Withdraw token with specified amount.
function withdrawToken(address _token, uint256 _amount) external
onlyAdmin {
    require(_amount != 0, "ERROR: Invalid amount");
    uint256 _curAmount = IERC20(_token).balanceOf(address(this));
    require(_curAmount >= _amount, "ERROR: Current balance is too
low");
    IERC20(_token).transfer(msg.sender, _amount);
}
///@notice Deposit token with specified amount.
function depositToken(address _token, uint256 _amount) external
onlyAdmin {
    require(isAcceptingToken(_token), "ERROR: Invalid token");
    require(_amount != 0, "ERROR: Invalid amount");
    IERC20(_token).transferFrom(msg.sender, address(this), _amount);
}
```

However, it uses *transfer()/transferFrom()* directly, instead of using a safe transfer library. There are hundreds of tokens who do not use the standard ERC20 signature and return **void**. Such tokens (USDT, BNB, etc.) would be incompatible with the Vault.

Recommended mitigation

Use the SafeERC20 library. Indeed, it has already been imported to Vault.

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

Team response

Fixed.

Mitigation review

All transfers have been fixed in Vault. However, StargatePlugin still uses unsafe transfers.

TRST-M-9 Vault does not have a way to withdraw native tokens

- **Category:** Leak of value issues
- **Source:** Vault.sol
- **Status:** Fixed + new issue

Description

The Vault sets the LayerZero fee refund address to itself:

```
/// @notice Report snapshot of the vault to the controller.
function reportSnapshot() public onlyBridge {
    MozBridge.Snapshot memory _snapshot = _takeSnapshot();
    MozBridge(mozBridge).reportSnapshot(_snapshot,
payable(address(this)));
}
```

However, there is no function to withdraw those funds, making them forever stuck in the vault only available for paying for future transactions.

Recommended mitigation

Add a native token withdrawal function.

Team response

Fixed.

Mitigation review

The fix includes a new *withdraw()* function. Its intention is to vacate any ETH stored in the controller and vaults.

```
function withdraw() public {
    // get the amount of Ether stored in this contract
    uint amount = address(this).balance;
    // send all Ether to owner
    // Owner can receive Ether since the address of owner is payable
    (bool success, ) = treasury.call{value: amount}("");
    require(success, "Controller: Failed to send Ether");
}
```

In fact, attackers can simply call *withdraw()* to make messaging fail due to lack of native tokens. This could be repeated in every block to make the system unusable.

TRST-M-10 MozBridge underestimates gas for sending of Moz messages

- **Category:** Gas-related issues
- **Source:** MozBridge.sol
- **Status:** Fixed

Description

The bridge calculates LayerZero fees for sending Mozaic messages using the function below:

```
function quoteLayerZeroFee(
    uint16 _chainId,
    uint16 _msgType,
    LzTxObj memory _lzTxParams
) public view returns (uint256 _nativeFee, uint256 _zroFee) {
    bytes memory payload = "";
    if (_msgType == TYPE_REPORT_SNAPSHOT) {
        payload = abi.encode(TYPE_REPORT_SNAPSHOT);
    }
    else if (_msgType == TYPE_REQUEST_SNAPSHOT) {
        payload = abi.encode(TYPE_REQUEST_SNAPSHOT);
    }
    else if (_msgType == TYPE_SWAP_REMOTE) {
        payload = abi.encode(TYPE_SWAP_REMOTE);
    }
    else if (_msgType == TYPE_STAKE_ASSETS) {
        payload = abi.encode(TYPE_STAKE_ASSETS);
    }
    else if (_msgType == TYPE_UNSTAKE_ASSETS) {
        payload = abi.encode(TYPE_UNSTAKE_ASSETS);
    }
    else if (_msgType == TYPE_REPORT_SETTLE) {
        payload = abi.encode(TYPE_REPORT_SETTLE);
    }
    else if (_msgType == TYPE_REQUEST_SETTLE) {
        payload = abi.encode(TYPE_REQUEST_SETTLE);
    }
    else {
        revert("MozBridge: unsupported function type");
    }

    bytes memory _adapterParams = _txParamBuilder(_chainId, _msgType,
_lzTxParams);
    return layerZeroEndpoint.estimateFees(_chainId, address(this),
payload, useLayerZeroToken, _adapterParams);
}
```

The issue is that the actual payload used for Mozaic messages is longer than the one calculated above. For example, REPORT_SNAPSHOT messages include a **Snapshot** structure.

```
struct Snapshot {
    uint256 depositRequestAmount;
    uint256 withdrawRequestAmountMLP;
    uint256 totalStablecoin;
    uint256 totalMozaicLp; // Mozaic "LP"
    uint8[] pluginIds;
    address[] rewardTokens;
    uint256[] amounts;
}
```

Undercalculation of gas fees will cause insufficient gas to be sent to the bridge, reverting the *send()* transaction.

Recommended mitigation

Error on the side of caution and estimate a larger than expected fee.

Team response

Fixed.

Mitigation review

The code now uses the correct payload for estimating fees.

TRST-M-11 No slippage protection for cross-chain swaps in StargatePlugin

- **Category:** MEV attacks
- **Source:** StargatePlugin.sol
- **Status:** Fixed

Description

The StargatePlugin calls StargateRouter's *swap()* function to do a cross-chain swap.

```
// Swaps
IStargateRouter(_router).swap(_dstChainId, _srcPoolId, _dstPoolId,
payable(address(this)), _amountLD, 0, IStargateRouter.lzTxObj(0, 0,
"0x"), abi.encodePacked(_to), bytes(""));
```

It will pass 0 as the minimum amount of tokens to receive. This pattern is vulnerable to sandwich attacks, where the fee or conversion rate is pumped to make the user receive hardly any tokens. In Layer Zero, the equilibrium fee can be manipulated to force such losses.

Recommended mitigation

Calculate accepted slippage off-chain, and pass it to the *_swapRemote()* function for validation.

Team response

Fixed.

Mitigation review

Affected function has been removed.

Low severity findings

TRST-L-1 Theoretical reentrancy attack when TYPE_MINT_BURN proposals are executed

- **Category:** Reentrancy issues
- **Source:** Multisig.sol
- **Status:** Fixed

Description

The senate can pass a proposal to mint or burn tokens.

```
if(proposals[_proposalId].actionType == TYPE_MINT_BURN) {
    (address _token, address _to, uint256 _amount, bool _flag) =
abi.decode(proposals[_proposalId].payload, (address, address,
uint256, bool));
    if(_flag) {
        IXMozToken(_token).mint(_amount, _to);
    } else {
        IXMozToken(_token).burn(_amount, _to);
    }
    proposals[_proposalId].executed = true;
}
```

Note that the proposal is only marked as executed at the end of execution, but execution is checked at the start of the function.

```
function execute(uint256 _proposalId) public onlyCouncil {
    require(proposals[_proposalId].executed == false, "Error:
Proposal already executed.");
    require(proposals[_proposalId].confirmation >= threshold, "Error:
Not enough confirmations.");
```

Interaction with tokens should generally be assumed to grant arbitrary call execution to users. If the *mint* or *burn()* calls call *execute()* again, the proposal will be executed twice, resulting in double the amount minted or burned. Specifically for XMoz, it is not anticipated to yield execution to the **to** address, so the threat remains theoretical.

Recommended mitigation

Follow the Check-Effects-Interactions design pattern, mark the function as executed at the start.

Team response

Fixed.

Mitigation review

The *execute()* function is now protected with the **nonReentrant** modifier.

TRST-L-2 XMozToken permits transfers from non-whitelisted addresses

- **Category:** Logical flaws
- **Source:** XMozToken.sol
- **Status:** Fixed + new issue

Description

The XMozToken is documented to forbid transfers except from whitelisted addresses or mints.

```
/**
* @dev Hook override to forbid transfers except from whitelisted
addresses and minting
*/
function _beforeTokenTransfer(address from, address to, uint256
/*amount*/) internal view override {
    require(from == address(0) || _transferWhitelist.contains(from)
|| _transferWhitelist.contains(to), "transfer: not allowed");
}
```

However, as can be seen, non-whitelisted users can still transfer tokens, so long as it is to whitelisted destinations.

Recommended mitigation

Remove the additional check in *_beforeTokenTransfer()*, or update the documentation accordingly.

Team response

Fixed.

Mitigation review

The check was changed as seen below:

```
function _beforeTokenTransfer(address from, address to, uint256
/*amount*/) internal view override {
    require(from == address(0) || to == address(0) || from == owner()
|| isTransferWhitelisted(from), "transfer: not allowed");
}
```

The original issue is fixed, however the **to == address(0)** check introduced a new major issue. It is used to allow the burning of tokens for the MozStaking contract. As a side-effect, it allows users to bridge the XMoz token (which is an OFTv2 token behind the covers). A user can bypass the transfer whitelist by bridging the asset and specifying any recipient.

TRST-L-3 XMozToken cannot be added to its own whitelist

- **Category:** Logical flaws
- **Source:** XMozToken.sol
- **Status:** Fixed

Description

By design, XMozToken should always be in the whitelist. However, *updateTransferWhitelist()* implementation forbids both removal and insertion of XMozToken to the whitelist.

```
function updateTransferWhitelist(address account, bool add) external
onlyMultiSigAdmin {
    require(account != address(this), "updateTransferWhitelist:
Cannot remove xMoz from whitelist");
    if(add) _transferWhitelist.add(account);
    else _transferWhitelist.remove(account);
    emit SetTransferWhitelist(account, add);
}
```

Recommended mitigation

Move the **require** statement into the **else** clause.

Team response

Fixed.

Mitigation review

The issue has been addressed by manually adding the contract to the transfer whitelist in the constructor.

TRST-L-4 Vault will fail to operate plugins with USDT due to lack of zero approval

- **Category:** ERC20 compatibility issues
- **Source:** Vault.sol
- **Status:** Open (Partial fix)

Description

Admins can trigger the *execute()* function to interact with plugins.

```
if(_actionType == IPlugin.ActionType.Stake) {
    (uint256 _amountLD, address _token) = abi.decode(_payload,
(uint256, address));
    IERC20(_token).approve(supportedPlugins[_pluginId].pluginAddr,
_amountLD);
} else if(_actionType == IPlugin.ActionType.SwapRemote) {
    (uint256 _amountLD, address _token, , ) = abi.decode(_payload,
(uint256, address, uint16, uint256));
    IERC20(_token).approve(supportedPlugins[_pluginId].pluginAddr,
_amountLD);
}
IPlugin(supportedPlugins[_pluginId].pluginAddr).execute(_actionType,
_payload);
```

Some tokens like USDT will revert when calling approve() when the previous allowance is not zero (to protect against a well-known double-allowance attack). Therefore, if the plugin does not consume the entire allowance, in the next *execute()* call the function will revert.

Recommended mitigation

Call *approve(token, 0)* before setting the desired approval.

Team response

Fixed.

Mitigation review

Similar to M-8, the issue has been fixed correctly in the Vault, but unsafe approvals remain in the StargatePlugin contract.

TRST-L-5 Controller doesn't initialize supported chains properly

- **Category:** Initialization flaws
- **Source:** Controller.sol
- **Status:** Acknowledged

Description

Controller manages a list of **supportedChainIds** where the protocol is deployed. The **mainChainId** is the base chain where controller is deployed. For settling and snapshotting, there is different handing in case the current chainID is **mainChainId**:

```
if(mainChainId == supportedChainIds[i]) {
    MozBridge.Snapshot memory _snapshot = mozBridge.takeSnapshot();
    _updateSnapshot(mainChainId, _snapshot);
} else {
    mozBridge.requestSnapshot(supportedChainIds[i], payable(master));
}
```

In fact, the **mainChainId** is not inserted to the list by default and needs to be manually added. For the protocol to function, it must be part of the list.

Recommended mitigation

During construction, add **mainChainId** to the list of supported chains.

Team response

Acknowledged.

Additional recommendations

Optimize gas usage in MozToken

In *getLockedPerAgreement()*, if the **elapsed time = vestPeriod**, it would be best to just set **lockedAmount = 0**.

```
if (((block.timestamp - vestStart) / 86400) <=
currentVest.vestPeriod) {
    unLockedAmount = currentVest.totalAmount * ((block.timestamp -
vestStart) / 86400) / currentVest.vestPeriod;
    lockedAmount = currentVest.totalAmount - unLockedAmount;
} else {
   lockedAmount = 0;
}
```

Theoretical reentrancy protection in MozToken

MozToken guards against transferring of locked tokens using *_beforeTokenTransfer()* hook. It is recommended to instead use the *_afterTokenTransfer()* hook, because if the transfer operation somehow reenters MozToken *transfer()*, the check would be outdated.

XMozToken should sanitize whitelist changes better

The *updateTransferWhitelist()* allows adding an account that is already in the whitelist, or removing one that isn't in it. Consider checking the return value of *add()/remove()* operations.

Cache invariants

In some parts of the codebase, external calls are used to get values at every invocation although they are not expected to ever change. For example, the *convertLDtoMD()* function:

```
function convertLDtoMD(address _token, uint256 _amountLD) public view
returns (uint256) {
    uint8 _localDecimals = IERC20Metadata(_token).decimals();
    if (MOZAIC_DECIMALS >= _localDecimals) {
        return _amountLD * (10**(MOZAIC_DECIMALS - _localDecimals));
    } else {
        return _amountLD / (10**(_localDecimals - MOZAIC_DECIMALS));
    }
}
```

The token's *decimals()* are fixed. Consider caching them for improved performance.

Define immutables

State variables that should never change during the lifetime of a contract should be defined as immutable. For example, the Vault's **chainId** is fixed. The use of immutables both improve readability and save loads from storage.

Emit events for state changes

Any important changes in state should be documented in events. Consider covering the transition of states in Controller and settlements in Vault.

Better sanitization in *removeToken()* of Vault

Currently, *removeToken()* allows removal of tokens that are not in the **acceptedTokens** list.

Better sanitization in *_getPoolIndexInFarming()* of StargatePlugin

If the call to *_getPool()* returns an address of zero, the plugin could confuse it as a valid found pool. It is best to revert in that scenario.

Use of "0x" for address strings

In several instances in StargatePlugin and MozBridge, "0x" is used for an empty address. It is recommended to use address(0) instead, to save gas costs and reduce confusion. In some cases, "0x" could be interpreted as a valid address.

Centralization risks

Admin is in control of the funds

A malicious or hacked admin account has many different ways to drain user's funds and take over governance permanently. This includes, but is not limited to:

1. Using *withdrawToken()* in Vault.sol
2. Setting trusted addresses like the bridge, LP token and plugins in Vault.sol
3. Changing plugin attributes and logic through *configPlugin()* in StargatePlugin.sol
4. Setting trusted remote bridges trough *setBridge()* in MozBridge

Admin can mint or burn fees

The Moz and XMoz tokens expose *mint()/burn()* interfaces that an admin can use arbitrarily.

Admin can set plugin fee to 100%

The admin can set Mozaic's cut from strategy earnings through *setFee()*, up to 100%.

```
function setFee(uint256 _mozaicFeeBP, uint256 _treasuryFeeBP)
external onlyOwner {
    require(_mozaicFeeBP <= BP_DENOMINATOR, "Stargate: fees > 100%");
    require(_treasuryFeeBP <= BP_DENOMINATOR, "Stargate: fees >
100%");
    mozaicFeeBP = _mozaicFeeBP;
    treasuryFeeBP = _treasuryFeeBP;
}
```

Systemic risks

Delivery of messages across chains

LayerZero is used to transport messages cross-chain. If a compromise of LayerZero take place, the worst case scenario may include forgeries of Mozaic messages, causing havoc and possibly loss of funds.

Plugins

Any funds held by external plugins are exposed to third-party risks. Currently, that comprises of Stargate centralization-related threats and security compromise potential.