Ion Protocol Security Review

Pashov Audit Group

Conducted by: unforgiven, Said, Shaka, santipu, Dan Ogurtsov, carrotsmuggler

July 10th 2024 - July 13th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Ion-Protocol/ion-boring-vault** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Ion

Ion Protocol is a decentralized money market built for staked and restaked assets. Borrowers can collateralize their yield-bearing staking assets to borrow WETH, and lenders can gain exposure to the boosted staking yield generated by borrower collateral.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes:

- d29fee34dcc0e6188688273df9ad0888003a589f
- 65678ddca45e33ebe1a49e368f12342e66788221

fixes review commit hash - 5a6e990d1dac7c4c0eda445e25ff7aabec8afbc0

Scope

The following smart contracts were in scope of the audit:

- `CrossChainLayerZeroTellerWithMultiAssetSupport`
- `CrossChainTellerBase`
- `OAppAuth`
- `OAppAuthCore`
- `OAppAuthReceiver`
- `OAppAuthSender`
- `TellerWithMultiAssetSupport`
- `BoringVaultCrossChainDepositor`
- `BoringVaultL2OFT`
- `BoringVaultOFTAdapter`

7. Executive Summary

Over the course of the security review, unforgiven, Said, Shaka, santipu, Dan Ogurtsov, carrotsmuggler engaged with Ion Protocol to review Ion. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Ion
Repository	https://github.com/Ion-Protocol/ion-boring-vault
Date	July 10th 2024 - July 13th 2024
Protocol Type	Lending Protocol

Findings Count

Severity	Amount
High	1
Medium	2
Low	9
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Users can bypass shareUnlockTime	High	Resolved
[M-01]	Missing check for supported asset	Medium	Resolved
[M-02]	Not checking allowMessagesFrom on the destination chain	Medium	Resolved
[L-01]	shareUnlockTime will impact existing shares	Low	Acknowledged
[L-02]	bridge does not check if the provided msg.value is enough for the fee	Low	Acknowledged
[L-03]	Lack of requiresAuth modifier	Low	Resolved
[L-04]	Vault is incompatible with tokens exhibiting "weird" traits	Low	Acknowledged
[L-05]	beforeTransfer is checked inside the bridge	Low	Acknowledged
[L-06]	Lack of minimum gas sent on bridging leads to a loss of shares	Low	Resolved
[L-07]	Extra and enforced options for lzReceive gas limit not handled properly	Low	Acknowledged
[L-08]	No way to revoke approval for unsupported assets	Low	Acknowledged
[L-09]	Bridge does not work if minting with native assets	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] Users can bypass `shareUnlockTime`

Severity

Impact: Medium

Likelihood: High

Description

When users call `depositAndBridge`, it will immediately transfer the token to the vault, then users will receive the minted shares and bridge it to the destination chain.

```
function depositAndBridge(
      ERC20depositAsset,
      uint256depositAmount,
      uint256minimumMint,
      BridgeDatacalldatadata
    ) external payable{
        uint shareAmount = _erc20Deposit
          (depositAsset, depositAmount, minimumMint, msg.sender);
        bridge(shareAmount, data);
    }
```

On the destination chain, it will then mint the shares to the destination receiver.

```
function _lzReceive(
        Origin calldata _origin,
        bytes32 _guid,
        bytes calldata payload,
        address,  // Executor address as specified by the OApp.
        bytes calldata  // Any extra data or options to trigger on receipt.
    ) internal override {
        // @audit - should call _afterPublicDeposit if it from depositAndBridge
        // Decode the payload to get the message
        (uint256 shareAmount, address receiver) = abi.decode(payload,
          (uint256,address));
>>>     vault.enter(address(0), ERC20(address(0)), 0, receiver, shareAmount);
    }
```

```
function receiveBridgeMessage
      (address receiver, uint256 shareMintAmount) external{
        if(msg.sender != address(messenger)){
            revert CrossChainOPTellerWithMultiAssetSupport_OnlyMessenger();
        }

        if(messenger.xDomainMessageSender() != peer){
            revert CrossChainOPTellerWithMultiAssetSupport_OnlyPeerAsSender();
        }

>>>     vault.enter(address(0), ERC20(address
  (0)), 0, receiver, shareMintAmount);
    }
```

This operation can be abused if the bridging time (from source to destination) is less than the `shareLockPeriod`. Users could make their shares transferable sooner than the `shareLockPeriod`.

Recommendations

Consider passing additional data or flag in the cross-chain operation, if it is called from `depositAndBridge`, consider calling `_afterPublicDeposit` for the `receiver` in the destination chain.

8.2. Medium Findings

[M-01] Missing check for supported asset

Severity

Impact: Low

Likelihood: High

Description

The `depositAndBridge` function currently lacks the necessary validation to verify that the deposit asset is supported by the Vault. This omission could allow an attacker to use a custom token as the deposit asset, misleading the Vault into minting shares that are not backed by approved assets.

```
function depositAndBridge(
  ERC20depositAsset,
  uint256depositAmount,
  uint256minimumMint,
  BridgeDatacalldatadata
) external payable{
    uint shareAmount = _erc20Deposit
      (depositAsset, depositAmount, minimumMint, msg.sender);
    bridge(shareAmount, data);
}
```

Currently, this attack may not be exploitable because it depends on some admin variables in the `AccountantWithRateProviders` contract. However, this bug could become worrying in some scenarios where the protocol has deprecated a supported asset but it has forgotten to remove the price feeds at `AccountantWithRateProviders`.

Recommendations

To rectify this issue, it is recommended to integrate a check in the `depositAndBridge` function to verify that the deposit asset is indeed supported by the Vault:

```
function depositAndBridge(
      ERC20depositAsset,
      uint256depositAmount,
      uint256minimumMint,
      BridgeDatacalldatadata
    ) external payable{
+       require(isSupported[depositAsset]);
        uint shareAmount = _erc20Deposit
          (depositAsset, depositAmount, minimumMint, msg.sender);
        bridge(shareAmount, data);
    }
```

[M-02] Not checking `allowMessagesFrom` on the destination chain

Severity

Impact: High

Likelihood: Low

Description

The Cross-chain Teller contract has configurations for each supported chain. They can separately configure whether to only receive messages (`allowMessagesFrom`), only send messages (`allowMessagesTo`), or have both configured as true or false.

```
function addChain(
        uint32 chainSelector,
        bool allowMessagesFrom,
        bool allowMessagesTo,
        address targetTeller,
        uint64 messageGasLimit
    ) external requiresAuth {
        if (allowMessagesTo && messageGasLimit == 0) {
            revert CrossChainTellerBase_ZeroMessageGasLimit();
        }
        selectorToChains[chainSelector] = Chain
          (allowMessagesFrom, allowMessagesTo, targetTeller, messageGasLimit);

        emit ChainAdded(
          chainSelector,
          allowMessagesFrom,
          allowMessagesTo,
          targetTeller,
          messageGasLimit
        );
    }
```

However, when receiving cross-chain message, it doesn't consider the `allowMessagesFrom` flag.

```
function _lzReceive(
        Origin calldata _origin,
        bytes32 _guid,
        bytes calldata payload,
        address,  // Executor address as specified by the OApp.
        bytes calldata  // Any extra data or options to trigger on receipt.
    ) internal override {
        // @audit - not checking "allowMessagesFrom"
        // Decode the payload to get the message
        (uint256 shareAmount, address receiver) = abi.decode(payload,
          (uint256,address));
        vault.enter(address(0), ERC20(address(0)), 0, receiver, shareAmount);
    }
```

```
function receiveBridgeMessage
      (address receiver, uint256 shareMintAmount) external{
        // @audit - not checking receive chain from?
        if(msg.sender != address(messenger)){
            revert CrossChainOPTellerWithMultiAssetSupport_OnlyMessenger();
        }

        if(messenger.xDomainMessageSender() != peer){
            revert CrossChainOPTellerWithMultiAssetSupport_OnlyPeerAsSender();
        }

        vault.enter(address(0), ERC20(address
          (0)), 0, receiver, shareMintAmount);
    }
```

This can potentially cause issues. For instance, if the supported chain configuration only allows sending messages to the destination chain (`allowMessagesTo` is true) but does not allow receiving messages from that chain (`allowMessagesFrom` is set to false). Another scenario is in the case of an emergency where the other chain has a malicious activity that needs to prevent receiving messages from that chain.

Recommendations

Check `allowMessagesFrom` when receiving a message from the other chain.

8.3. Low Findings

[L-01] `shareUnlockTime` will impact existing shares

When a user deposits via Teller, it will update their `shareUnlockTime`, preventing them from transferring their shares until `shareUnlockTime` has passed. However, this design will also impact their existing shares, which should not be affected by the `shareUnlockTime`, potentially disrupting the user experience. Consider redesigning the unlock implementation to only impact the deposited shares instead of affecting all of the user's shares.

[L-02] `bridge` does not check if the provided `msg.value` is enough for the fee

`CrossChainLayerZeroTellerWithMultiAssetSupport` has `_quote`, which can be used to calculate the fee required for the operation using best practices from LayerZero. However, when `_bridge` is called, it doesn't check if the provided `msg.value` is at least equal to the `_quote` estimation. Consider adding an additional check to ensure that the provided `msg.value` is enough to cover the fee.

[L-03] Lack of `requiresAuth` modifier

While this function is meant to be publicly callable, it is not consistent with the deposit implementation inside `TellerWithMultiAssetSupport`, where all functions still have `requiresAuth`. Consider adding the modifier inside the `bridge` function as well.

[L-04] Vault is incompatible with tokens exhibiting "weird" traits

Some tokens that adhere to the ERC-20 standard exhibit unusual characteristics, such as rebasing or fee-on-transfer mechanisms. Integrating such tokens into the Vault can disrupt its accounting system and lead to financial losses for its users.

For instance, integrating a rebasing token like `stETH` could compromise the Vault's accounting because it does not account for the automatic balance adjustments (`stETH` balance increases every block). Similarly, incorporating fee-on-transfer tokens like `STA` or `PAXG` could cause discrepancies since the Vault does not anticipate receiving fewer tokens than requested during a `transferFrom` call.

To prevent accounting issues, it is advisable not to integrate tokens with these complex traits into the Vault. This approach will help maintain the integrity and accuracy of the Vault's accounting system.

[L-05] `beforeTransfer` is checked inside the bridge

When users call `depositAndBridge`, they will immediately deposit the token to the vault, and the shares minted will be immediately provided to the bridge function to bridge the shares to the destination chain.

```
function depositAndBridge(
      ERC20depositAsset,
      uint256depositAmount,
      uint256minimumMint,
      BridgeDatacalldatadata
    ) external payable{
        uint shareAmount = _erc20Deposit
          (depositAsset, depositAmount, minimumMint, msg.sender);
>>>     bridge(shareAmount, data);
    }
```

```
function bridge(
      uint256shareAmount,
      BridgeDatacalldatadata
    ) public payable returns(bytes32 messageId
        if(isPaused) revert TellerWithMultiAssetSupport__Paused();
        if
          (!selectorToChains[data.chainSelector].allowMessagesTo) revert CrossChainTel

        if
          (data.messageGas > selectorToChains[data.chainSelector].messageGasLimit){
            revert CrossChainTellerBase_GasLimitExceeded();
        }

        // Since shares are directly burned, call `beforeTransfer` to enforce
        // before transfer hooks.
        // @audit - should not be checked if called from deposit and bridge
>>>     beforeTransfer(msg.sender);

        // Burn shares from sender
        vault.exit(address(0), ERC20(address(0)), 0, msg.sender, shareAmount);

        messageId = _bridge(shareAmount, data);
        _afterBridge(shareAmount, data, messageId);
    }
```

However, the bridge function will always check `beforeTransfer` regardless of whether it is called directly or from `depositAndBridge`. It should not check the user's `shareUnlockTime` since it is only bridging shares from the deposit operation inside `depositAndBridge`.

```
function beforeTransfer(address from) public view {
        if
          (shareUnlockTime[from] >= block.timestamp) revert TellerWithMultiAssetSuppor
    }
```

This check could cause an unexpected revert if previous users deposit assets via Teller and have not yet passed the unlock time.

Consider updating the bridge function, if it is called from `depositAndBridge`, skip the `beforeTransfer` check.

[L-06] Lack of minimum gas sent on bridging leads to a loss of shares

In cross-chain operations, it's crucial to send an adequate amount of gas to ensure the success of transactions on the destination chain. The `CrossChainTellerBase` contract currently validates that the gas amount does not exceed the maximum set by protocol administrators. However, it lacks a check for the minimum required gas.

```
if(data.messageGas > selectorToChains[data.chainSelector].messageGasLimit){
    revert CrossChainTellerBase_GasLimitExceeded();
}
```

When a user carelessly sends a cross-chain transaction with too little `messageGas`, the transaction may fail due to insufficient gas, causing the user to lose the shares involved in the bridging process.

To mitigate this issue, it is recommended that a minimumMessageGas variable be introduced within the `Chain` struct. This variable would define the minimum gas required for a transaction to succeed across different EVM chains. Setting a higher minimum is preferable since unused gas will be refunded to the user. This addition will help prevent transactions from failing due to low gas limits.

```
struct Chain{
      bool allowMessagesFrom;
      bool allowMessagesTo;
      address targetTeller;
      uint64 messageGasLimit;
+     uint64 minimumMessageGas
  }

    function bridge(
      uint256shareAmount,
      BridgeDatacalldatadata
    ) public payable returns(bytes32 messageId
        // ...
        if
          (data.messageGas > selectorToChains[data.chainSelector].messageGasLimit){
            revert CrossChainTellerBase_GasLimitExceeded();
        }

+       if
+ (data.messageGas < selectorToChains[data.chainSelector].minimumMessageGas){
+           revert CrossChainTellerBase_GasTooLow();
+        }

        // ...
    }
```

Moreover, the LayerZero documentation recommends setting the "Enforced Options" to guarantee that the message sent from a source has sufficient gas to be executed on the destination chain, effectively preventing failures due to gas-related issues.

[L-07] Extra and enforced options for `lzReceive` gas limit not handled properly

`BoringVaultCrossChainDepositor.deposit()` receives `gasLimit` as a parameter. This value represents the amount of gas to be used in the `lzReceive` call by the Executor on the destination chain and is passed in `extraOptions` of the `SendParam` struct sent to `BoringVaultOFTAdapter.send`.

```
File: BoringVaultCrossChainDepositor.sol

    function deposit(
        ERC20 depositAsset,
        uint256 depositAmount,
        uint256 minimumMint,
        address bridgeRecipient,
 @>     uint128 gasLimit
    ) external payable returns (uint256 shares) {
 @>     bytes memory options = OptionsBuilder.newOptions
   ().addExecutorLzReceiveOption(gasLimit, 0);

        depositAsset.safeTransferFrom(msg.sender, address(this), depositAmount);
        shares = teller.deposit(depositAsset, depositAmount, minimumMint);

        if (shares < minimumMint) revert BelowMinimumMint(shares, minimumMint);

        SendParam memory sendParam = SendParam({
            dstEid: dstEid,
            to: bytes32(uint256(uint160(bridgeRecipient))),
            amountLD: shares,
            minAmountLD: shares,
 @>         extraOptions: options,
            composeMsg: "",
            oftCmd: ""
        });

        MessagingFee memory messagingFee = MessagingFee
          ({nativeFee: msg.value, lzTokenFee: 0});

        oftAdapter.send{value: msg.value}(
          {_sendParam:sendParam,
          _fee:messagingFee,
          _refundAddress:msg.sender}
        );
    }
```

On the other hand, the gas limit for the executor in the destination chain can also be set by the owner as an enforce option, which is recommended in the integration checklist.

> It is recommended to implement and set Enforced Options to ensure that the user of the OApp or OFT covers a predetermined amount of gas for every transaction. This setup guarantees that the message sent from a source has sufficient gas to be executed on the destination chain, effectively preventing failures due to gas-related issues. Please see Enforced Options for more information.

There are two possibilities:

1. The protocol sets the enforced options

In this case, the caller of `deposit` can be charged twice for the gas limit. We can see the following warnings in the documentation:

> CAUTION: When setting enforcedOptions, try not to unintentionally pass a duplicate _options argument to extraOptions. Passing identical _options in both enforcedOptions and extraOptions will cause the protocol to charge the caller twice on the source chain, because LayerZero interprets duplicate _options as two separate requests for gas.
>
> CAUTION: As outlined above, decide on whether you need an application wide option via enforcedOptions or a call specific option using extraOptions. Be specific in what _options you use for both parameters, as your transactions will reflect the exact settings you implement.

2. The protocol does not set the enforced options

In this case, the caller of `BoringVaultCrossChainDepositor.deposit` or `BoringVaultOFTAdapter.send` might set as `gasLimit` a value that is not enough for the executor to execute the transaction on the destination chain and thus, not being able to receive the tokens on the destination chain, that have already been locked on the source chain.

This is the scenario that seems more likely, as neither the deployment scripts, nor the tests set the enforced options.

Remove the `gasLimit` parameter from `BoringVaultCrossChainDepositor.deposit` and use the enforced options to set the gas limit for the executor on the destination chain.

[L-08] No way to revoke approval for unsupported assets

In `BoringVaultCrossChainDepositor` anyone can set max approvals if the token is supported on Teller:

```
function maxApprove(ERC20 depositAsset) external {
        if (!teller.isSupported(depositAsset)) revert AssetNotSupported();
        depositAsset.approve(address(boringVault), type(uint256).max);
    }
```

But Teller can either add supported assets or remove them. For removed, assets consider revoking approvals.

[L-09] Bridge does not work if minting with native assets

Description

The teller contract accepts native assets (`msg.value`) in order to mint vault shares.

```
if (address(depositAsset) == NATIVE) {
            if (msg.value == 0) revert TellerWithMultiAssetSupport__ZeroAssets
              ();
            nativeWrapper.deposit{value: msg.value}();
            depositAmount = msg.value;
            shares = depositAmount.mulDivDown
              (ONE_SHARE, accountant.getRateInQuoteSafe(nativeWrapper));
            if
              (shares < minimumMint) revert TellerWithMultiAssetSupport__MinimumMintNo
            // `from` is address(this) since user already sent value.
            nativeWrapper.safeApprove(address(vault), depositAmount);
            vault.enter(address
              (this), nativeWrapper, depositAmount, msg.sender, shares);
```

However this is no longer possible with the cross-chain bridge. This is because the `teller.deposit()` call is not passed with a msg.value.

```
shares = teller.deposit(depositAsset, depositAmount, minimumMint);
```

Furthermore, all the native assets passed in is used for the crosschain call. So no assets can be used to mint the shares from the vault.

Mitigation recommendation

If native assets based mints are to be supported, pass in an extra `mintWithAmount`, and use only that amount in the `teller.deposit{value:`

`mintWithAmount}()` call. In all the other places, replace `msg.value` with the remaining balance of eth.