Redacted Pirex Eth Mode Security Review

Security Review

Cantina Managed review by:
Optimum, Lead Security Researcher
Defsec, Security Researcher
Gjaldon, Junior Security Researcher

June 9, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

The Redacted ecosystem is a product suite of smart contracts empowering on-chain liquidity, governance, and cash flow for DeFi protocols.

From Apr 9th to Apr 15th the Cantina team conducted a review of dinero-pirex-eth/src/mode on commit hash 7b3c723a. The team identified a total of **14** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 5
- Low Risk: 2
- Gas Optimizations: 3
- Informational: 3

3 Findings

3.1 High Risk

3.1.1 Non-zero withdrawal penalty in `AutoPxEth` can lead to insolvency for `ModeEthLockbox`

Severity: High Risk

Context: AutoPxEth.sol#L387

Description: When there is a non-zero withdrawal penalty for `AutoPxEth`, this can lead to insolvency in `ModeEthLockbox`. This is because the withdrawal penalty is applied during withdrawal or redemption but not during deposits in `autoPxEth`:

```
function previewWithdraw(
    uint256 assets
) public view override returns (uint256) {
    // ... snip ...
    // Factor in additional shares to fulfill withdrawal if user is not the last to withdraw
    return
        (_totalSupply == 0 || _totalSupply - shares == 0)
            ? shares
            : (shares * FEE_DENOMINATOR) /
                (FEE_DENOMINATOR - withdrawalPenalty);
}
```

```
function previewDeposit(uint256 assets) public view virtual returns (uint256) {
    return convertToShares(assets);
}

function convertToShares(uint256 assets) public view virtual returns (uint256) {
    uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is non-zero.

    return supply == 0 ? assets : assets.mulDivDown(supply, totalAssets());
}
```

Consider the following scenario:

1. Alice deposits 100 pxETH into `ModeEthLockbox`. Given a `totalSupply` of 1000 and `totalAssets` of 2000, 50 apxEth shares will be minted to the `ModeEthLockbox`.
2. In `ModeEth`, the deposit of 100 pxEth will mint 50 modeEth shares to Alice given `totalShares` of 1000 and `totalStaked` of 2000.
3. Alice withdraws the 100 pxEth from `ModeEth` which in turn withdraws 100 pxEth from `AutoPxEth` in `ModeEthLockbox`.

```
uint256 postFeeAmount = _calculateWithdrawalAmount(
    _amount,
    _assetsPerShare
);

autoPxEth.withdraw(postFeeAmount, _receiver, address(this));
```

4. The withdrawal from `AutoPxEth` burns 55 apxEth shares since it is calculated as `convertToShares(assets) / 100% - withdrawal penalty %`. Note that only 50 apxEth shares were minted to the `ModeEthLockbox` when Alice deposited 100 pxETH.

In effect, more apxEth shares will be burned on every withdrawal than are minted on every deposit. This leads to insolvency in `ModeEthLockbox` since not every `ModeEth` shares can be redeemed. Later withdrawers will have their withdrawal of modeEth succeed, but it will revert when `ModeEthLockbox` processes the LayerZero message for the withdrawal.

The withdrawal penalty is 30_000 by default.

Recommendation: Consider deducting the withdrawal penalty when calculating the pxEth withdrawal amount in `ModeEthLockbox`.

Redacted: `withdrawPenalty` is being deducted only when withdrawing. We've replaced `assetsPerShare` for `convertToAssets`. See commit f330a819.

3.2 Medium Risk

3.2.1 In `ModeEth`, depositors can avoid the fee on earnings by withdrawing before `rebase` is called

Severity: Medium Risk

Context: ModeEth.sol#L209

Description: The `rebase` function is in charge of updating the `assetsPerShare` ratio and calculate the collectible amount for treasury fees for the `ModeEth` contract on the Mode network. Fees are a portion of the earnings made by rewards from the `apxEth` balance that the `ModeEthLockbox` contract holds and are allocated as `ModeEth` tokens (shares) by calling `_mintShares`. In the current version of the code, the `ModeEthLockbox.rebase` sends the absolute amount which from the portion of `treasuryFee` should be taken by the `ModeEth._lzReceive` function.

Depositors that wish to avoid these fees can spot the `rebase` transaction on the mempool and call `ModeEth.withdrawModeEth` right before. In the case of a single depositor, it will cause the upcoming call to `_lzReceive` (with a `_messageType = MESSAGE_TYPE_REBASE`) to revert for an underflow / division by zero on this line:

```
shares = fee.mulDivDown(totalShares, _totalAssets() - fee);
```

In the case of multiple depositors, the fee that should be paid will be levied on the other depositors causing them to effectively pay more than the `treasuryFee` rate.

3.2.2 Rebase fails when `assetsPerShare` decreases

Severity: Medium Risk

Context: ModeEthLockbox.sol#L374

Description: The `ModeEthLockbox` allows sending a rebase instruction to `ModeEth`. This is how `ModeEth` can distribute yield from `autoPxEth` interest accruals in Mainnet.

This `rebase()` function implicitly expects that `assetsPerShare` is a monotonically increasing value.

```
function rebase(bytes calldata _options) external payable nonReentrant {
    uint256 assetsPerShare = autoPxEth.assetsPerShare();
    uint256 assetsIncrease = assetsPerShare - lastAssetsPerShare;
    // ... snip ...
}
```

However, the `assetsPerShare` can decrease when the withdrawal penalty increases.

See AutoPxEth.sol#L379-L394:

```
function previewRedeem(
    uint256 shares
) public view override returns (uint256) {
    // Calculate assets based on a user's % ownership of vault shares
    uint256 assets = convertToAssets(shares);

    uint256 _totalSupply = totalSupply;

    // Calculate a penalty - zero if user is the last to withdraw.
    uint256 penalty = (_totalSupply == 0 || _totalSupply - shares == 0)
        ? 0
        : assets.mulDivUp(withdrawalPenalty, FEE_DENOMINATOR); // Round up the penalty in favour of the
↪   protocol.

    // Redeemable amount is the post-penalty amount
    return assets - penalty;
}
```

Note that `autoPxEth.assetsPerShare()` calls `previewRedeem()` internally.

In effect, rebase will fail due to an underflow (`assetsPerShare - lastAssetsPerShare`) until the ratio of assets:shares increases enough to cover the increase in withdrawal penalty.

Recommendation: Consider computing for the `assetsPerShare` without subtracting the withdrawal penalty or changing the rebase logic to make it work for negative rebases.

Redacted: To avoid this issue, we replaced `assetsPerShare` in favor of `convertToAssets(1e18)`. See commit f330a819.

3.2.3 Missing `assetsPerShare` value in `quoteDeposit` function

Severity: Medium Risk

Context: ModeEthLockbox.sol#L421

Description: The `quoteDeposit` function is used to quote the gas cost for deposit messages. However, the payload encoding in this function does not include the `assetsPerShare` value, which represents the exchange rate between the share token (apxEth) and the underlying asset (pxEth) at the time of the deposit.

The correct payload for the deposit message should include the `assetsPerShare` value, as it is necessary for accurately calculating the number of shares to be minted for the deposited assets.

By not including the `assetsPerShare` value in the payload, the quote calculation may be inaccurate, as it will not consider the correct exchange rate between apxEth and pxEth at the time of the deposit.

```
/**
 * @notice Quote gas cost for deposit messages
 * @param  _receiver  address  The recipient of the deposit in Mode.
 * @param  _amount    uint256  Deposit amount.
 * @param  _options   bytes    Additional options for the message.
 */
function quoteDeposit(
    address _receiver,
    uint256 _amount,
    bytes calldata _options
) external view returns (MessagingFee memory msgFee) {
    bytes memory payload = abi.encode(
        MESSAGE_TYPE_DEPOSIT,
        _amount, // Use amount directly for quote
        uint256(0),
        _receiver
    );

    bytes memory combinedOptions = combineOptions(MODE_EID, 0, _options);

    return _quote(MODE_EID, payload, combinedOptions, false);
}
```

Recommendation: Include the `assetsPerShare` value in the payload encoding of the `quoteDeposit` function. One way to achieve this is by using the `autoPxEth.convertToAssets(1e18)` function, which returns the current `assetsPerShare` value.

Here's how the `quoteDeposit` function can be modified:

```
function quoteDeposit(
    address _receiver,
    uint256 _amount,
    bytes calldata _options
) external view returns (MessagingFee memory msgFee) {
    bytes memory payload = abi.encode(
        MESSAGE_TYPE_DEPOSIT,
        _amount, // Use amount directly for quote
        autoPxEth.convertToAssets(1e18), // Include the current assetsPerShare value
        _receiver
    );
    bytes memory combinedOptions = combineOptions(MODE_EID, 0, _options);
    return _quote(MODE_EID, payload, combinedOptions, false);
}
```

Redacted: Fixed in commit 6712542b.

3.2.4 Incorrect asset value assumption in `quoteWithdraw` function

Severity: Medium Risk

Context: ModeEth.sol#L274

Description: The `quoteWithdraw` function is used to quote the gas cost for withdrawal messages. However, the payload encoding in this function sends the `totalStaked` value instead of the `lastAssetsPerShare` value.

The correct payload for the withdrawal message should include the `lastAssetsPerShare` value, which represents the exchange rate between the share token (apxEth) and the underlying asset (pxEth) at the time of the last successful rebase operation.

By sending the `totalStaked` value instead of `lastAssetsPerShare`, the quote calculation will be inaccurate, as it will not consider the correct exchange rate.

```
/**
 * @notice Quote gas cost for withdrawal messages
 * @param _receiver  address  The recipient of the withdrawal on Mainnet.
 * @param _amount    uint256  The withdrawal amount.
 * @param _options   bytes    Additional options for the message.
 */
function quoteWithdraw(
    address _receiver,
    uint256 _amount,
    bytes calldata _options
) external view returns (MessagingFee memory msgFee) {
    bytes memory _payload = abi.encode(
        MESSAGE_TYPE_WITHDRAW,
        _amount,
        totalStaked,
        _receiver
    );

    bytes memory _combinedOptions = combineOptions(
        MAINNET_EID,
        0,
        _options
    );

    return _quote(MAINNET_EID, _payload, _combinedOptions, false);
}
```


Recommendation: Replace the `totalStaked` value in the payload encoding with the `lastAssetsPerShare` value, as shown below:

```
bytes memory _payload = abi.encode(
    MESSAGE_TYPE_WITHDRAW,
    _amount,
    lastAssetsPerShare, // Use lastAssetsPerShare instead of totalStaked
    _receiver
);
```

Redacted: Fixed in commit 6712542b.

3.2.5 Loss of excess fees for contract senders

Severity: Medium Risk

Context: ModeEthLockbox.sol#L248

Description: In the `_lzSend` function, the excess fees (native and LZ token fees) sent to the LayerZero endpoint are refunded to the `_refundAddress` provided as a parameter. However, this refund mechanism may not work as intended if the `msg.sender` is a contract that is not deployed on the destination chain.

When a contract initiates the `depositPxEth` & `depositEth` & `depositApxEth` & `rebase` function, it becomes the `msg.sender`, and its address is passed as the `_refundAddress` to the `_lzSend` function. If this contract is not deployed on the destination chain, any excess fees refunded to this address will be lost since the contract does not exist on the destination chain to receive the refund.

```
MessagingReceipt memory msgReceipt = _lzSend(
    MODE_EID,
    payload,
    combinedOptions,
    MessagingFee(msg.value - _amount, 0), // Taken from remaining amount after deducting deposit amount
    payable(msg.sender)
);
```

Recommendation: To mitigate this issue, introduce an additional parameter in the functions to explicitly specify the refund address. This refund address should be an address that is guaranteed to exist on the destination chain, ensuring that excess fees are refunded correctly.

Here's an example of how the `depositPxEth` function could be modified:

```
function depositPxEth(
    address _receiver,
    uint256 _amount,
    bytes calldata _options,
    address _refundAddress
) external payable nonReentrant {
    // ... (existing code)

    // Call _lzSend with the provided _refundAddress
    MessagingReceipt memory msgReceipt = _lzSend(
        _dstEid,
        payload,
        combinedOptions,
        MessagingFee(msg.value, 0),
        _refundAddress
    );

    // ... (existing code)
}
```

Redacted: Fixed in commit 9ddb8822.

3.3 Low Risk

3.3.1 Increase in withdrawal penalty leads to incorrect adjustments to `assetsPerShare`

Severity: Low Risk

Context: ModeEthLockbox.sol#L474

Description: An adjustment is done to the `assetsPerShare` when there is a change in the withdrawal penalty when the withdrawal amount is calculated in `ModeEthLockbox`:

```
function _calculateWithdrawalAmount(
    uint256 _amount,
    uint256 _assetsPerShare
) internal view returns (uint256) {
    uint256 currentAssetsPerShare = autoPxEth.assetsPerShare();
    uint256 currentWithdrawalPenalty = autoPxEth.withdrawalPenalty();

    int256 penaltyDifference = int256(currentWithdrawalPenalty) -
        int256(lastWithdrawalPenalty);

    uint256 adjustedFactor = uint256(1e6 - penaltyDifference.abs());

    // adjust _amount and _assetsPerShare with the penalty difference
    if (penaltyDifference != 0) {
        bool isPenaltyPositive = penaltyDifference > 0;
        _amount = _amount.mulDivDown(
            isPenaltyPositive ? adjustedFactor : 1e6,
            isPenaltyPositive ? 1e6 : adjustedFactor
        );
        _assetsPerShare = _assetsPerShare.mulDivDown(
            isPenaltyPositive ? adjustedFactor : 1e6,
            isPenaltyPositive ? 1e6 : adjustedFactor
        );
    }
```

However, the adjustment calculation is incorrect when there is an increase in the withdrawal penalty. Consider the following scenario:

- Assets per share without the withdrawal penalty applied is at 1000.
- In `ModeEthLockbox`, the `assetsPerShare` is 900 since the withdrawal penalty is already 10%.
- In `ModeEth`, the `assetsPerShare` is 950 since the withdrawal penalty was still at 5% when the withdrawal was executed.
- Since there is a 5% increase in withdrawal penalty, the adjusted factor is 95%.
- The adjusted `_assetsPerShare` will be `950 * 0.95 ~= 902` instead of 900.

This leads to a smaller amount of assets being withdrawn, although only by dust amounts and no major issues caused by this have been identified.

Recommendation: Consider relying solely on `amount * currentAssetsPerShare / assetsPerShare` to adjust the withdrawal amount and remove the `penaltyDifference != 0` block.

Redacted: Need to sync and adjust `withdrawPenalty` was removed. Fixed in commit f330a819.

3.3.2 The usage of `balanceOf` is discouraged in `ModeEthLockbox`

Severity: Low Risk

Context: ModeEthLockbox.sol#L380, ModeEthLockbox.sol#L486, ModeEthLockbox.sol#L501

Description: `ModeEthLockbox` is making use of the `ERC20.balanceOf` function in three different instances:

- ModeEthLockbox.sol#L378-L382:

```
// amount increased by pxEth rewards accrued
uint256 amount = assetsIncrease.mulDivDown(
    autoPxEth.balanceOf(address(this)),
    1e18
);
```

- ModeEthLockbox.sol#L485-L487:

```
uint256 totalAssets = autoPxEth.previewRedeem(
    autoPxEth.balanceOf(address(this))
);
```

- ModeEthLockbox.sol#L501:

```
uint256 totalShares = autoPxEth.balanceOf(address(this));
```

As known, anyone can "donate" `autoPxEth` tokens to the `ModeEthLockbox` which might cause the normal behavior to diverge a bit, although we could not find any major issue around it.

Recommendation: Consider maintaining a storage variable in the `ModeEthLockbox` contract to track the total `apxEth` shares held by the `ModeEthLockbox` contract. This variable should increase upon deposits and decrease upon withdrawals, and should be used instead of `balanceOf` to better harden the system.

Redacted: Fixed in eab28fa5.

3.4 Gas Optimization

3.4.1 Reentrancy guards can be implemented in transient storage

Severity: Gas Optimization

Context: ModeEth.sol#L19, ModeEthLockbox.sol#L23

Description: Both `ModeEthLockbox` and `ModeEth` use the `ReentrancyGuard` library to ensure re-entrant calls will cause a transaction revert. Solidity 0.8.24 allows access to two newly introduced opcodes `TSTORE` and `TLOAD` that will make re-entrancy guards cheaper.

Recommendation: Consider using the `ReentrancyGuardTransient` library as a replacement. Please keep in mind that it will require bumping the solidity version and that this library only works on networks where EIP-1153 is available. For more information refer to transient-storage.

3.4.2 Consider reverting the call in case `amount = 0` in `ModeEthLockbox.rebase`

Severity: Gas Optimization

Context: ModeEthLockbox.sol#L372

Description/Recommendation: The `rebase` function is in charge of updating the `assetsPerShare` ratio and calculate the collectible amount for fees for the `ModeEth` contract on the Mode network. As we can see in this code snippet:

```
function rebase(bytes calldata _options) external payable nonReentrant {
    uint256 assetsPerShare = autoPxEth.assetsPerShare();
    uint256 assetsIncrease = assetsPerShare - lastAssetsPerShare;

    if (assetsIncrease == 0) revert Errors.InvalidAmount();

    // amount increased by pxEth rewards accrued
    uint256 amount = assetsIncrease.mulDivDown(
        autoPxEth.balanceOf(address(this)),
        1e18
    );

    bytes memory payload = abi.encode(
        MESSAGE_TYPE_REBASE,
        amount,
        assetsPerShare,
        address(0)
    );
// ...
```

In case the amount is 0, this function should revert as it should be considered a no-op and might save gas for the keeper bot.

Redacted: Fixed in commit a4536e05.

3.4.3 Shares calculation can be simpler and more gas-efficient

Severity: Gas Optimization

Context: ModeEthLockbox.sol#L224

Description: When depositing ETH to the `ModeEthLockbox` contract, the amount of `apxEth` shares received is computed via the `authPxEth.convertToShares()` function.

```
(uint256 postFeeAmount, ) = pirexEth.deposit{value: _amount}(
    address(this),
    true
);

uint256 shares = autoPxEth.convertToShares(postFeeAmount);
```

This can be replaced with a simpler and more gas-efficient calculation using the contract's `autoPxEth` balances before and after depositing to the `PirexEth` vault.

Recommendation: The computation for `autoPxEth` shares can be simplified when depositing ETH to `ModeEthLockbox`. Below is an example of how this modification could look like:

```
uint256 preBalance = autoPxEth.balanceOf(address(this));
(uint256 postFeeAmount, ) = pirexEth.deposit{value: _amount}(
    address(this),
    true
);

uint256 shares = autoPxEth.balanceOf(address(this)) - preBalance;
```

Redacted: Fixed in commit 1890d721.

3.5 Informational

3.5.1 Consider implementing a shared abstract contract for constant variables in `ModeEth` and `ModeEthLockbox`

Severity: Informational

Context: ModeEthLockbox.sol#L45-L57, ModeEth.sol#L43-L55

Description/Recommendation: Both contracts are using constants such as `MESSAGE_TYPE_DEPOSIT`, `MESSAGE_TYPE_WITHDRAW`, `MESSAGE_TYPE_REBASE` which can be shared in a common abstract contract to ensure minimal unintentional errors of wrong values assigned.

Redacted: Fixed in commit 7069ab17.

3.5.2 `_lastAssetsPerShare` declaration and assignment can be moved out of the `if`/`else` blocks in `ModeEth._lzReceive`

Severity: Informational

Context: ModeEth.sol#L180

Description/Recommendation: `_lastAssetsPerShare` is being used in two different code paths but is being declared twice, as we can see in this code snippet:

```
// ...
if (_messageType == MESSAGE_TYPE_DEPOSIT) {
    uint256 shares = totalShares == 0
        ? _amount
        : convertToShares(_amount);
    uint256 _totalStaked = totalStaked;
    uint256 _lastAssetsPerShare = lastAssetsPerShare;

    _mintShares(_receiver, shares);

    // update total staked
    _lastAssetsPerShare == 0
        ? totalStaked = _totalStaked + _amount
        : totalStaked =
        _totalStaked.mulDivDown(_assetsPerShare, _lastAssetsPerShare) +
        _amount;

    // update last assets per share
    lastAssetsPerShare = _assetsPerShare;

    emit Deposit(_guid, _receiver, shares, _amount);
} else if (_messageType == MESSAGE_TYPE_REBASE) {
    // update total staked
    totalStaked = totalStaked.mulDivDown(
        _assetsPerShare,
        lastAssetsPerShare
    );

    // update last assets per share
    lastAssetsPerShare = _assetsPerShare;
// ...
```

To improve code readability and make maintenance easier, consider moving the declaration of `_lastAssetsPerShare` outside of the if/else blocks instead, the `MESSAGE_TYPE_REBASE` code path should use `_lastAssetsPerShare` instead of `lastAssetsPerShare` and the update to `lastAssetsPerShare` can be moved to the end of the function:

```
// ...
uint256 _lastAssetsPerShare = lastAssetsPerShare;

if (_messageType == MESSAGE_TYPE_DEPOSIT) {
    uint256 shares = totalShares == 0
        ? _amount
        : convertToShares(_amount);
    uint256 _totalStaked = totalStaked;

    _mintShares(_receiver, shares);

    // update total staked
    _lastAssetsPerShare == 0
        ? totalStaked = _totalStaked + _amount
        : totalStaked =
        _totalStaked.mulDivDown(_assetsPerShare, _lastAssetsPerShare) +
        _amount;

    emit Deposit(_guid, _receiver, shares, _amount);
} else if (_messageType == MESSAGE_TYPE_REBASE) {
     // update total staked
    totalStaked = totalStaked.mulDivDown(
        _assetsPerShare,
        _lastAssetsPerShare
    );

    uint256 fee = _amount.mulDivDown(treasuryFee, FEE_DENOMINATOR);

    uint256 shares;
    if (fee > 0) {
        shares = fee.mulDivDown(totalShares, _totalAssets() - fee);

        _mintShares(treasury, shares);
    }

    emit Rebase(_guid, treasury, _assetsPerShare, _amount, fee, shares);
} else {
    revert Errors.NotAllowed();
}
// update last assets per share
lastAssetsPerShare = _assetsPerShare;
```

Redacted: Fixed in commit b0f72f41.

3.5.3 Lack of pausable feature for cross-chain deposits

Severity: Informational

Context: ModeEthLockbox.sol#L263

Description: The `depositPxEth` function is responsible for accepting pxEth deposits and minting apxEth shares to be stored in the vault, as well as sending a message to the Mode chain. However, there is currently no mechanism to pause or disable this function, which could be problematic in case of emergencies or unexpected situations involving cross-chain communication.

In scenarios where the cross-chain communication between the mainnet and Mode chains is disrupted or compromised, it may be necessary to temporarily pause new deposits to prevent potential issues or losses. Without a pausable feature, the contract remains vulnerable to accepting new deposits, even in situations where it may be advisable to halt these operations.

```
function depositEth(
    address _receiver,
    address _refundAddress,
    uint256 _amount,
    bytes calldata _options
)
```

Recommendation: Introduce a pausable mechanism for the `deposit` functions, allowing authorized parties (e.g., contract owners or administrators) to temporarily disable new deposits when necessary. This can be achieved by implementing a modifier or a state variable that controls the pausability of the function.

Redacted: Fixed in commit 602baf06.