Redacted Cartel Security Review

Auditors

Rajeev, Lead Security Researcher

0x52, Lead Security Researcher

Slowfi, Security Researcher

Ayeslick, Junior Security Researcher

Report prepared by: Pablo Misirov

September 19, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Pirex is designed to simplify the management of staked tokens. It provides auto-compounding, yield management, and gives users access to liquid wrappers for staked tokens. By expanding the utility of staked tokens, Pirex allows users to access unique features and yield-earning opportunities.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of dinero-pirex-eth according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 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

4 Executive Summary

Over the course of 8 days in total, Redacted Cartel engaged with Spearbit to review the dinero-pirex-eth protocol. In this period of time a total of **35** issues were found.

Summary

Project Name	Redacted Cartel
Repository	dinero-pirex-eth
Commit	77bf93...c395
Type of Project	Staking, DeFi
Audit Timeline	July 24 to Aug 2
Two week fix period	Aug 2 - Aug 16
Extension period	2 days
Extension commit	13ed4f...0048

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	6	6	0
Medium Risk	7	6	1
Low Risk	14	14	0
Gas Optimizations	2	2	0
Informational	5	4	1
Total	**35**	**33**	**2**

5 Findings

5.1 Critical Risk

5.1.1 `PirexEthValidators.receive()` handles execution-layer rewards incorrectly

Severity: Critical Risk

Context: PirexEthValidators.sol#L131-L133

Description: `PirexEthValidators` is meant to handle Execution-Layer rewards (MEV and Tips) along with Consensus-Layer rewards (Proposer, Attestation and Sync Committee). Validators running Proposer-Builder-Separation (PBS) software, such as Flashbots MEV-Boost, maximize their staking reward by selling blockspace to an open market of builders. Such builders use a standard EVM transfer to send MEV to the intended fee recipient, which would effectively be `PirexEthValidators` in this case. However, the current `receive()` function would treat these MEV ETH payments as a donation towards pending deposits rather than rewards to be harvested by minting `pxEth`.

Recommendation: Remove `_addPendingDeposit(msg.value)` from `receive()`.

Redacted: PR#13.

Spearbit: Reviewed that PR#13 fixes the issue as recommended.

5.2 High Risk

5.2.1 Slashing a non `Withdrawable` validator accounts the wrong amount for `outstandingRedemptions`

Severity: High Risk

Context: PirexEthValidators.sol#L412

Description: Fixes on Commit 52925f0 introduce a new bug. The `slashValidator()` function adds to `outstandingRedemptions` the full deposit size every time a validator is slashed. However if the validator is staking then there is no need to increment `outstandingRedemptions` because no `pxETH` has been redeemed against it.

Recommendation: Consider not adding the `DEPOSIT_SIZE` amount if the validator is staking.

Redacted: Commit 63450d4.

Spearbit: Reviewed that Commit 63450d4 fixes the issue as recommended.

5.2.2 Slashing and validator penalties break critical protocol invariant of 1:1 `pxEth` peg

Severity: High Risk

Context: PirexEthValidators.sol#L280-L294, PirexEthValidators.sol#L380-L400

Description: A slashing event which affects protocol validators, or validator penalties due to incorrect or missing attestations will reduce the amount of the validator ETH balances. The implementation however does not burn a corresponding amount of `pxEth` after a slashing event or after using buffer funds while topping up in the case of penalties.

Impact: This breaks a critical protocol invariant of 1:1 `pxEth` peg, which means that there will be more `pxEth` in the protocol than what can be redeemed against available ETH.

Medium likelihood (Slashing/Penalties is possible) + High impact (Critical protocol invariant violation) = High severity.

Recommendation: Burn appropriate amounts of `pxEth` in the slashing flow and if/when the top-up flow uses buffer funds. The latter aspect is also addressed in *`topUpStake()`'s use of `buffer` leads to partial loss of user funds*.

Redacted: PR#30, Commit 52925f0.

Spearbit: Reviewed that PR#30 and changes made until Commit 52925f0 fix the issue as per the recommendation via the use of newly introduced concept of burner accounts. Slashing now optionally uses buffer funds to

compensate for the slashing loss. Both slashing and top-ups using buffer funds now burn a corresponding amount of `pxEth` from governance approved/managed burner accounts.

5.2.3 Operations on initialized validators using indices may impact unexpected validators

Severity: High Risk

Context: PirexEthValidators.sol#L321-L326, PirexEthValidators.sol#L332-L336, PirexEthValidators.sol#L343-L352

Description: Any function which uses a queue index parameter and therefore relies on the underlying queue being unchanged is broken due to TOCTOU (classic race-condition). This is presumably implemented this way to prevent traversing the queue looking for a pubkey (O[n] vs O[1] for unordered).

Similar to *`slashValidator()` can remove an incorrect validator from `_stakingValidators` leading to loss/lock of funds*, the Governance operations on initialized validators that use indices or expect certain validators at certain positions in the queue, i.e. `swapInitializedValidator()`, `popInitializedValidator()` and `removeInitializedValidator()` will impact the incorrect validator(s) if there were any other operations such as a deposit affecting the indices in-between the index determination and the function execution.

Impact: The wrong validators would get swapped, popped or removed leading to incorrect protocol state and loss/lock of user funds.

Medium likelihood (deposits are expected to happen at any time) + High impact (protocol state corruption and loss/lock of funds) = High severity.

Recommendation: One mitigation would be for Governance to execute these queue-modifying functions only after pausing the protocol deposits, in which case they should check `depositEtherPaused` for these functions.

Redacted: PR#31.

Spearbit: Reviewed that PR#31 fixes the issue as recommended by enforcing a modifier that checks for `depositEtherPaused` on all these add/swap/pop/remove/clear functions.

5.2.4 `redeemWithUpxEth()` and `redeemWithPxEth()` redeem ETH to incorrect address leading to potential loss/lock of funds

Severity: High Risk

Context: PirexEth.sol#L222C14-L241, PirexEth.sol#L254-L283

Description: Both `redeemWithUpxEth()` and `redeemWithPxEth()` accept a `_receiver` parameter as the address of the ETH receiver. However, they incorrectly send back ETH to `msg.sender`.

Impact: If the callers of `redeemWithUpxEth()` and `redeemWithPxEth()` specify a different receiver address they will not receive ETH at that address. Depending on the scenario, e.g. use of relayers, this will lead to loss/lock of user funds.

Medium likelihood (depends on `_receiver` != `msg.sender`) + High impact (loss/lock of funds) = High severity.

Recommendation: Change `msg.sender` to `_receiver`.

Redacted: PR#10.

Spearbit: Reviewed that PR#10 fixes the issue by changing the `msg.sender` to `_receiver` in both functions.

5.2.5 `slashValidator()` can remove an incorrect validator from `_stakingValidators` leading to loss/lock of funds

Severity: High Risk

Context: PirexEthValidators.sol#L280-L290, PirexEthValidators.sol#L547-L557, PirexEth.sol#L222-L231

Description: `slashValidator()` takes a `_removeIndex` of the validator to be slashed within the protocol to keep it in sync with the validator slashing observed in the Consensus layer. However, this assumes that `_stakingValidators` queue is unchanged from when `_removeIndex` was determined and the time of `slashValidator()` execution. This will not be the case if a `deposit()`, `initiateRedemption()` or another `slashValidator()` were to be executed in-between during that time window, which would have changed the indices of `_stakingValidators` queue elements.

Impact: This will cause an incorrect validator to be removed from the queue which will be different from what should be considered slashed i.e. `_pubKey` in the protocol. The validator which should have been actually slashed would still be considered as being in the staking state within the protocol and would be processed in the initiate redemption flow where its `_pubKey` would be associated within `batchIdToValidator`. This `_pubkey` can however never be dissolved via `dissolveValidator()` because it belongs to the slashed validator which cannot perform a voluntary exit, which means that `status[_pubKey]` can never be `DataTypes.ValidatorStatus.Dissolved`. The staker who was minted `upxEth` corresponding to this `_pubKey`'s `batchId` will therefore always have their `redeemWithUpxEth()` revert at the check `status[batchIdToValidator[_tokenId]] != DataTypes.ValidatorStatus.Dissolved` leading to a loss/lock of their funds.

Medium likelihood (slashing mixed with the other actions is likely) + High impact (loss/lock of funds from being unable to redeem back ETH using `upxEth`) = High severity.

Recommendation: Consider having the `removeOrdered()` and `removeUnordered()` functions take `_pubKey` as a parameter and require that to be the same as `_removedPubkey`, which would prevent incorrect removals but require re-execution with correct `_removeIndex` under such scenarios. Alternatively, these functions could return the `_removedPubkey` to allow checking against `_pubKey`.

Redacted: PR#28.

Spearbit: Reviewed that PR#28 fixes the issue (as per the alternative recommendation) by having the `ValidatorQueue` functions `removeOrdered()` and `removeUnordered()` return the `removedPubKey` and asserting that it matches the `_pubKey` of the staking validator meant to be slashed in `slashValidator()`.

5.2.6 Incorrect access control prevents keepers from fulfilling their role responsibilities

Severity: High Risk

Context: PirexEthValidators.sol#L280-L285, PirexEthValidators.sol#L380-L386, Specification

Description: Keepers in the protocol are special roles whose responsibilities include: "*`KEEPER_ROLE` - harvest rewards. Update status when a validator is slashed , top up the validator stake when active balance goes below effective balance*", as per the specification.

However, functions `slashValidator()` and `topUpStake()` incorrectly enforce the `onlyRole(GOVERNANCE_ROLE)` modifier. Governance role is expected to be different and generally more privileged than the Keeper roles in protocols where Governance is typically the protocol multisig or the DAO, while Keepers are semi-trusted protocol incentivized actors. This incorrect access control enforcement will await Governance to execute Keeper responsibilities which may be unexpected and therefore never happen. This will also prevent Keepers from fulfilling their role responsibilities which could lead to critical protocol accounting going out of sync, e.g. `pxEth` not pegged 1:1 with `ETH`, when validators are slashed or top-ups are required.

High likelihood (Keepers are locked out of their roles) + Medium impact (Governance has to step in for Keepers) = High severity.

Recommendation: Change access control on `slashValidator()` and `topUpStake()` to `onlyRole(KEEPER_ROLE)`.

Redacted: PR#14.

Spearbit: Reviewed that PR#14 fixes the issue by changing the access control permission to `KEEPER_ROLE` on both functions.

5.3 Medium Risk

5.3.1 Full redeem not possible if validator is slashed during exit

Severity: Medium Risk

Context: PirexEthValidators.sol#L395-L411, PirexEth.sol#L294

Description: The fixes in PR#30 introduced a new issue that makes it impossible to fully claim the ETH funds corresponding to redeemed `upxEth` if a validator is slashed during exit. Whenever a validator is in `Withdrawable` state, the `upxEth` minted for it is already issued. However if the validator is slashed while exiting (which is possible, see validator lifecycle), the contract receives only the ETH amount that remains after slashing which will be less than the redeemed `upxEth` amount. In such a scenario, users will be unable to fully claim the ETH corresponding to their `upxEth`.

Impact: Partial loss of user funds in this scenario.

Low likelihood (Slashing while exiting) + High impact (Partial loss of user funds) = Medium severity.

Recommendation: Consider reimplementing accounting mechanisms that control the amount users should claim when a validator is slashed during exit.

Redacted: Commit 52925f0.

Spearbit: Reviewed that Commit 52925f0 resolves the issue. The fix removes global state variables `totalPxEthAmount`, `totalDepositBackFromBeaconChain` and `totalEthRedeemedWithUpxEth` while introducing `pendingWithdrawal` and `outstandingRedemptions`. These new global state variables now hold values that allow correct accounting.

5.3.2 Centralization risk in the protocol can cause considerable harm

Severity: Medium Risk

Context: DineroERC20.sol#L41, DineroERC20.sol#L57, PxEth.sol#L30, PirexEth.sol#L94, PirexEthValidators.sol#L203, PirexEthValidators.sol#L285, PirexEthValidators.sol#L346,

Description: Privileged roles (e.g. `GOVERNANCE_ROLE`, `KEEPER_ROLE`, `MINTER_ROLE`, `BURNER_ROLE`, `OPERATOR_ROLE`, `DEFAULT_ADMIN_ROLE`) in the protocol can cause considerable harm if they are compromised or abused. They can arbitrarily change/affect the participating validators, update contract addresses, affect minting/burning of `pxEth`, pause deposits and cause other unexpected protocol changes via offchain actions, which can lead to lock/loss of user/protocol funds.

Low likelihood + High impact = Medium Severity.

Recommendation: Ensure that privileged roles are backed by a reasonable multisig or a token-based governance, and place their authorized functions behind a timelock.

Redacted: Acknowledged. Majority (if not all) will be under msig wallets and/or under sole direct control of the msig. Timelock (outside of ownership transfer mechanism) will not be implemented at the current state.

Spearbit: Acknowledged.

5.3.3 Unsynchronized offchain components will cause incorrect accounting of `totalDepositBackFromBeaconChain` and other unexpected states

Severity: Medium Risk

Context: PirexEthValidators.sol#L263-L270, PirexEthValidators.sol#L280-L293, PirexEthValidators.sol#L417-L441

Description: The current implementation of the protocol relies on an offchain component to update the `totalDepositBackFromBeaconChain` state variable in `PirexEthValidators.sol`. This variable accounts for the ETH released from validators when being slashed or exiting the network. This same contract also receives the Execution-Layer and Consensus-Layer rewards. Additionally, an offchain component is responsible for calling the `harvest()` function to distribute the rewards. The `harvest()` function calculates the rewards amount by subtracting the `totalDepositBackFromBeaconChain` amount from the current balance of the contract.

Given the current architecture, if the offchain components fail to synchronize their different function calls in the correct order, the ETH obtained back from a validator can be distributed as rewards. It also allows other unexpected state transitions but with less or no impact on user funds.

Recommendation: Consider implementing the following redesign: 1) Receive rewards and released ETH on different addresses 2) Make functions `dissolveValidator()` and `slashValidator()` payable and check that `msg.value` corresponds with expected amounts 3) Update `totalDepositBackFromBeaconChain` state variable appropriately 4) Finally, implement an offchain component that transfers the ETH to the `PirexEth.sol` contract's corresponding functions. The rewards can be directly sent as ETH transfer or to the `harvest()` function.

This redesign mitigates potential synchronization issues leading to unexpected states in the protocol.

Redacted: PR#30.

Spearbit: PR#30 partially solved the issue by introducing a separate contract that receives the rewards and the ETH back from the validators. However the initial fix introduced some other potential issues. These issues where resolved on Commit 63450d4 that fixed the issue completely.

The fix introduced the RewardRecipient.sol contract. This contract receives the rewards as well as the released ETH from the validators. The `RewardRecipient.sol` contains `dissolveValidator()`, `slashValidator()` and `harvest()` functions. These functions interact with their corresponding homologous functions of the `PirexEth.sol` contract. The `dissolveValidator()` can be called from the `OracleAdapter.sol` contract. On the other hand, the `slashValidator()` and `harvest()` functions can only be called by the `KEEPER` role. The three functions now send a fixed amount of ETH to the `PirexEth.sol` contract specified as an input parameter.

On the accounting logic side, the state variable `totalDepositBackFromBeaconChain` of `PirexEthValidators.sol` has been removed. Now the contract implements global state variables `pendingWithdrawal` and `outstandingRedemptions`. Also, the contract now uses `burnerAccounts`, which are specific accounts created by governance that holds `pxEth`, to help burning the required amounts of `pxEth` to maintain the peg under certain conditions.

5.3.4 A validator slashed after being associated with a redeem will lead to lock/loss of user's staked ETH

Severity: Medium Risk

Context: PirexEth.sol#L228-L231, PirexEthValidators.sol#L540-L554, PirexEthValidators.sol#L292

Description: It is possible for the validator `batchIdToValidator[_tokenId]]` to be slashed after a voluntary exit is triggered via `initiateRedemption()`. If so, such a slashed validator will have its ETH released but its status will be `Slashed` and not `Dissolved`, which will prevent `redeemWithUpxEth(_tokenId,..)` executing beyond the check `status[batchIdToValidator[_tokenId]] != DataTypes.ValidatorStatus.Dissolved` and therefore result in locked-and-loss-of staked ETH for the user.

The time period from the `RequestValidatorExit` emission to the validator successfully exiting depends on multiple factors including any offchain delays to signal a voluntary exit (by submitting a voluntary exit message to the beacon chain) and the exit delay itself which is "*a minimum 5 epochs (32 minutes) for a validator to exit after initiating a voluntary exit. This number can be much higher depending on how many other validators are queued to exit.*" If the validator is slashed during this time period for any reason, the associated `upxEth` cannot be redeemed.

From this reference: "*It is possible for a validator to be slashed while it is exiting and even after it has exited (but before the time its funds can be withdrawn). This stops validators from cheating and then avoiding punishment by exiting through the usual mechanism before their cheating is noticed.*", which describes the scenario where a dishonest validator would trigger a voluntary exit before performing a slashing condition.

Low likelihood (slashing is rare but possible) + High impact (lock/loss of user funds) = Medium severity.

Recommendation: Consider allowing redeeming even if the validator is slashed, along with appropriate changes to the slashing flow/logic.

Redacted: PR#30.

Spearbit: As part of the significant redesigning/refactoring in PR#30, `redeemWithUpxEth()` now allows redeeming if the validator status is either Dissolved or Slashed, which fixes this issue. The slashing flow/logic has also been significantly updated to allow slashing of validators in `Withdrawable` state among other things.

5.3.5 `pendingDeposit` getting too large may prevent subsequent deposits because of block gas limit

Severity: Medium Risk

Context: PirexEthValidators.sol#L450-L494, PirexEthValidators.sol#L501-L522, PirexEthValidators.sol#L364-L369

Description: If the `pendingDeposit` gets very large because of e.g. a large deposit or many pending deposits activated upon adding newly initialized validators, the `while` loop in `_deposit()` may exceed block gas limit, revert and therefore prevent any subsequent deposits from happening in the protocol.

Impact: Protocol deposits always revert subsequently and protocol stalls.

Medium likelihood (`pendingDeposit` needs to get large enough for loop to hit block gas limit) + Medium impact (protocol stalls) = Medium severity.

Recommendation: Consider parameterising the loop iterations in `_deposit()` (e.g. change to `for` loop) which can be set to a reasonable number especially for deposits via `depositPrivileged()` by Keepers. This will allows Keepers to resolve such a scenario by clearing the deposit backlog in batches.

Redacted: PR#23.

Spearbit: Reviewed that PR#23 fixes the issue by adding a `maxProcessedValidatorCount` (initialized to 20 and modifiable by `GOVERNANCE_ROLE`) that controls the number of validators deposited in `_deposit()` i.e. loop iterations.

5.3.6 `topUpStake()`'s use of `buffer` leads to partial loss of user funds

Severity: Medium Risk

Context: PirexEthValidators.sol#L377-L400

Description: The protocol allows Keepers (not Governance; see *Incorrect access control prevents keepers from fulfilling their role responsibilities*) to top up ETH for validators if their current balance drops below effective balance due to inactivity leaks. Given that validators are maintained by the protocol governance/team, any inactivity leaks and consequently required top-ups is their responsibility, which means that top-ups should be funded from the protocol treasury. However, `topUpStake()` has an option for Keepers to fund top-ups using `buffer` funds which is collected from user deposits.

Impact: `topUpStake()`'s optional use of `buffer` funds leads to partial loss of user deposits towards top-ups. This will lead to a corresponding depeg of the minted `pxEth` unless more ETH (equivalent to that lost due to inactivity leaks) is later introduced into the protocol.

Medium likelihood (Inactivity leaks due to validator downtime is likely) + Medium impact (partial loss of user funds) = Medium severity.

Recommendation: Remove the option for `topUpStake()` to use `buffer` funds for top-up and only allow top-ups with newly sent ETH via `msg.value`.

Redacted: PR#29, Commit 52925f0.

Spearbit: Reviewed that PR#29 fixes the issue as per the recommendation. Upon further changes as of Commit 52925f0, `topUpStake()` reintroduces and `slashValidator()` introduces the use of `buffer` funds for covering the losses but burn an equivalent amount of `pxEth` from governance-owned burner accounts, which effectively covers penalty costs (without penalising users) upon using buffer funds.

5.3.7 A compromised validator can front-run a deposit to register their withdrawal credentials for stealing subsequent deposits

Severity: Medium Risk

Context: PirexEthValidators.sol#L468-L480, Lido Vulnerability Mitigations, Lido Vulnerability Response

Description: The protocol could be susceptible to a deposit front-running vulnerability by compromised validators, which is described in the Lido vulnerability report as follows: "*The exploit is based on the fact that, as per the Ethereum consensus layer specification, the validator public key is associated with the withdrawal credentials (WC) on the first valid deposit that uses the public key. Subsequent deposits will use the WC from the first deposit even if another WC are specified.*"

Impact: Compromised validators can make an initial deposit with their WC and use that to withdraw funds of subsequent deposits from the protocol because the specified `withdrawalCredentials` in those calls is ignored.

Low likelihood (protocol controlled validators need to be compromised) + High impact (Loss of deposited funds) = Medium severity.

Recommendation: One approach to mitigating this is via pre-deposits to register the correct protocol WC for pubkeys of all initialized validators. Evaluate the mitigations suggested in Lido Vulnerability Mitigations.

Redacted: PR#30.

Spearbit: As part of the significant redesigning/refactoring in PR#30, the notion of `preDepositAmount` is introduced which is noted as: "// Deposit that a validator must do prior to adding to initialized validator queue". This is used to register the correct protocol WC (`RewardRecipient`) and is deducted while spinning up validators during deposits and a corresponding amount of `pxEth` is minted to the validator's receiver, which should mitigate the issue as recommended.

5.4 Low Risk

5.4.1 A stale value of `maxBufferSize` is used in deposits

Severity: Low Risk

Context: PirexEth.sol#L164-L183

Description: As a mitigation to *"An instantly redeeming staker can grief another by front-running to delay their redemptions"* makes the value of `maxBufferSize` dynamically dependent on the amount of `pxEth` minted. The `deposit()` function however uses a stale `maxBufferSize` value in `_addPendingDeposit()` because the minting of `pxEth` to the user,which updates the value of `maxBufferSize`, is done later. Note this was not a concern earlier because `maxBufferSize` was a constant value.

Recommendation: Consider moving `_addPendingDeposit()` to the end of the deposit function.

Redacted: Commit 054776c.

Spearbit: Reviewed that Commit 054776c resolves the issue as per the recommendation.

5.4.2 Accidentally locked ETH or non-protocol ERC20 tokens cannot be recovered in an emergency

Severity: Low Risk

Context: `PirexEthValidators.sol`

Description: Protocols holding ETH/ERC20 typically implement an emergency mechanism to recover ETH and non-protocol ERC20 tokens if/when they run into an exploit/unexpected situation. For e.g., if there are any latent issues in the reward/vault or other ETH/ERC20 flows which leads to these tokens getting exploited/stuck, protocol can pause, recover and remediate.

Recommendation: Consider implementing an emergency mechanism to allow governance to recover ETH or non-protocol ERC20 tokens.

Redacted: Commit 5d47c17, Commit 0087ac6, Commit 6472ada.

Spearbit: Reviewed that Commit 5d47c17, Commit 0087ac6, Commit 6472ada introduce emergency withdraw function as recommended.

5.4.3 Lack of indexed parameters in events will affect offchain searching/filtering

Severity: Low Risk

Context: PirexEth.sol#L36-L38, Events

Description: Events `InitiateRedemption`, `RedeemWithUpxEth` and `RedeemWithPxEth` do not use the `indexed` attribute on their key address parameters. This will add them to the data part of the log instead of the special data structure known as “topics”, which will make their searching/filtering inefficient.

Recommendation: Add `indexed` attribute to all key event parameters.

Redacted: PR#7.

Spearbit: Fixed as recommended.

5.4.4 Protocol uses OZ `AccessControl` instead of the newer safer extension

Severity: Low Risk

Context: OracleAdapter.sol#L9, DineroERC20.sol#L10, PirexEthValidators.sol#L19, ERC1155Solmate.sol#L27

Description: OpenZeppelin v4.9 has introduced a safer extension `AccessControlDefaultAdminRules` which implements, according to their announcement: "*To promote security best practices, we have implemented a set of AccessControl admin rules in code, which developers can use out of the box to secure their permissioning setup. The new system ensures that only one account and role can act as admin, with a two-step transfer process to ensure the default admin is never lost. There is also a configurable delay between admin transfers to reduce the risk of contract theft.*" However, protocol components use OZ `AccessControl` instead of the newer safer extension.

Impact: Protocol components are susceptible to the risks of OZ `AccessControl` which are mitigated by its safer extension `AccessControlDefaultAdminRules`.

Recommendation: Upgrade to `AccessControlDefaultAdminRules` as implemented in OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/extensions/AccessControlDefaultAdminRules.sol.

Redacted: PR#21.

Spearbit: Reviewed that PR#21 fixes the issue as per recommendation. An additional `_initialDelay` is passed to constructors as required for this extension. The zero-address check on `initialDefaultAdmin` is now delegated to the constructor of `AccessControlDefaultAdminRules`.

5.4.5 Single-step ownership transfer is risky

Severity: Low Risk

Context: PirexFees.sol#L12, AutoPxEth.sol#L14, Ownable2Step.sol

Description: `PirexFees` uses OpenZeppelin's `Ownable` while `AutoPxEth` uses Solmate's `Owned`. Both these only support single-step ownership transfers which are risky and allow accidental transfers to incorrect addresses.

Recommendation: Consider using OpenZeppelin's `Ownable2Step` which enforces a two-step ownership change allowing for mistakes to be corrected.

Redacted: PR#20.

Spearbit: Reviewed that PR#20 fixes the issue as per recommendation.

5.4.6 An instantly redeeming staker can grief another by front-running to delay their redemptions

Severity: Low Risk

Context: PirexEth.sol#L270, PirexEthValidators.sol#L39-L40

Description: An instantly redeeming staker can grief another by front-running to potentially make the `buffer` amount insufficient and therefore revert subsequent instant redeems. This will require redeems to either wait for `buffer` to be recapitalized via deposits or choose the much slower alternative via `initiateRedemption()` and `redeemWithUpxEth()`, which requires voluntarily exiting validators and waiting for their ETH to be deposited to the protocol before redeeming with `upxEth`.

Recommendation: The `maxBufferSize` should be configured to be sufficiently large to reduce this likelihood.

Redacted: PR#30.

Spearbit: Reviewed that PR#30 fixes the issue as recommended by dynamically adjusting `maxBufferSize` to be equal to `maxBufferSizePct` of `pxEth.totalSupply()` i.e. "Percentage from pxEth total supply will be allocated to max buffer size." This is updated during `pxEth` mints/burns in `_mintPxEth()` and `_burnPxEth()`.

5.4.7 Missing zero-address check may burn redeemed ETH

Severity: Low Risk

Context: PirexEth.sol#L241 PirexEth.sol#L283

Description: `redeemWithUpxEth()` and `redeemWithPxEth()` allow the caller to specify a `_receiver` address as the destination for the redeemed ETH. Note that the current implementation incorrectly uses `msg.sender` as the redeeming destination (see *`redeemWithUpxEth()` and `redeemWithPxEth()` redeem ETH to incorrect address leading to potential loss/lock of funds*). However, there is no zero-address check on the `_receiver` address which means that a user accidentally calling these functions with a zero-address `_receiver` will have their redeemed ETH burned.

Recommendation: Add a zero-address check as a precautionary measure to prevent users from losing their redeemed ETH.

Redacted: PR#10.

Spearbit: Reviewed that PR#10 fixes the issue as recommended. It also adds a similar zero-address check in `initiateRedemption` for early reverts in such scenarios.

5.4.8 Events emitted with incorrect/incomplete values will affect offchain monitoring and tooling

Severity: Low Risk

Context: PirexEth.sol#L197C14-L197C32, PirexEth.sol#L286, PirexEth.sol#L36-L38

Description: Events `InitiateRedemption` and `RedeemWithPxEth` are emitted only with the asset values without including the corresponding `postFeeAmount` values. This will reduce the visibility of these important protocol accounting values for any offchain monitoring and tooling. The event declaration `event RedeemWithPxEth(uint256 postFeeAmount, address _receiver);` even indicates that `RedeemWithPxEth` is meant to emit the `postFeeAmount` amount.

Recommendation: Add/change these events to emit both the asset and `postFeeAmount` values.

Redacted: PR#11.

Spearbit: Reviewed that PR#11 fixes the issue as recommended.

5.4.9 Staker funds could become dependent on other redemptions and remain locked for an indeterminate amount of time

Severity: Low Risk

Context: PirexEthValidators.sol#L560-L562

Description: When `totalPxEthAmount` is less than `DEPOSIT_SIZE` and therefore not sufficient to trigger a voluntary exit of a staking validator within the protocol, the `upxEth` minted to the receiver cannot be redeemed (no associated validator `_pubKey` for its `batchId`) until the total outstanding redemptions `totalPxEthAmount` again exceeds `DEPOSIT_SIZE`. This means that depending on the order of initiating redemptions, a staker could be left holding such `upxEth` that cannot be redeemed until other stakers sufficiently redeem via this flow as well. If other stakers continue to instantly redeem (via `redeemWithPxEth()`) then this staker's funds will be locked for an indeterminate amount of time.

Impact: Staker funds could become dependent on other redemption flows and remain locked until outstanding redemption amount again exceeds `DEPOSIT_SIZE`.

Recommendation: Consider a user specified boolean parameter for `initiateRedemption()` which allows the protocol to either continue with the existing behavior or revert in such a scenario. This will give users an option of attempting the alternative instant redemption (presumably with higher fee) if this flow reverts.

Redacted: PR#32.

Spearbit: Reviewed that PR#32 fixes the issue as recommended by adding a boolean parameter `_shouldTriggerValidatorExit` to `initiateRedemption()` which, when true, reverts any redemptions without corresponding validator exits.

5.4.10 Any excess ETH sent by Keepers for top-up is lost to protocol

Severity: Low Risk

Context: PirexEthValidators.sol#L400-L402

Description: The protocol allows Keepers (not Governance; see *Incorrect access control prevents keepers from fulfilling their role responsibilities*) to top up ETH for validators if their current balance drops below effective balance due to inactivity leaks. When more ETH than specified in the parameter `_topUpAmount` is sent via `msg.value`, only the `_topUpAmount` is deposited to the beacon chain for the specified validator but any excess ETH sent is not refunded back to the Keepers.

Impact: Any excess ETH (`msg.value - _topUpAmount`) is not refunded back to the Keepers but gets used within the protocol. This leads to a loss for the Keepers.

Low likelihood (Keepers should accidentally send excess ETH) + Medium impact (loss of excess ETH) = Low severity.

Recommendation: Prevent excess ETH from being sent by changing check to `if (msg.value != _topUpAmount)`.

Redacted: The `topUpAmount` argument is removed , however only amount required to top up validator is passed in the `value` field when keeper make a call to `topUpStake`. Fixed in PR#29.

Spearbit: Reviewed that PR#29 fixes the issue by sending all the `msg.value` to the validator.

5.4.11 Accidentally re-adding a validator to `_initializedValidators` will prevent future deposits

Severity: Low Risk

Context: PirexEthValidators.sol#L465-L466

Description: A used/deposited validator accidentally re-added to `_initializedValidators` via `addInitialized-Validator()` will revert when chosen for a deposit and therefore prevent any subsequent deposits in the protocol until such a validator is removed by governance.

Recommendation: Add a check `require(status[_pubKey] == DataTypes.ValidatorStatus.None)` before adding initialized validators to the protocol in `addInitializedValidator()` and `addInitializedValidators()`.

Redacted: PR#27.

Spearbit: Reviewed that PR#27 fixes the issue as recommended.

5.4.12 Missing events for critical privileged operations

Severity: Low Risk

Context: PirexEthValidators.sol#L263-L271, PirexEthValidators.sol#L280-L294, PirexEthValidators.sol#L380-L412

Description: Critical operations such as those access controlled by privileged protocol roles should emit events for transparency and monitoring. Some privileged functions such as `dissolveValidator()`, `slashValidator()` and `topUpStake()` are missing event emissions.

Recommendation: Consider adding appropriate events for critical privileged operations.

Redacted: PR#26.

Spearbit: Reviewed that PR#26 fixes the issue by emitting events with appropriate information, following the recommendation.

5.4.13 Using the `DEFAULT_ADMIN_ROLE` for `setContract()` and `setPirexEth()` is risky

Severity: Low Risk

Context: PirexEthValidators.sol#L200-L203, PirexEthValidators.sol#L125

Description: `setContract()` is a privileged function used to set/change all the critical contract addresses in the protocol of `PxEth`, `UpxEth`, `AutoPxEth` and `OracleAdapter`. However, the access control enforced on this function is `onlyRole(DEFAULT_ADMIN_ROLE)` instead of `onlyRole(GOVERNANCE_ROLE)`.

`setPirexEth()` is a privileged function used to set/change the `PirexEth` address within the `OracleAdapter`. However, the access control enforced on this function is `onlyRole(DEFAULT_ADMIN_ROLE)` instead of `onlyRole(GOVERNANCE_ROLE)`.

`DEFAULT_ADMIN_ROLE` is set to the `_admin` parameter in constructor. It is not specified if this will be the same as and as strongly secured as `GOVERNANCE_ROLE`. If the Governance role happens to be the protocol multisig or evolves to a DAO in future while `DEFAULT_ADMIN_ROLE` happens to be an EOA, then this is risky for the protocol because of: 1) Reduced security guarantees of `DEFAULT_ADMIN_ROLE` and 2) Unnecessary additional role to access control critical protocol contracts.

Recommendation: Change access control of `setContract()` to `onlyRole(GOVERNANCE_ROLE)`.

Redacted: PR#24.

Spearbit: Reviewed that PR#24 fixes the issue by changing the `setContract` access control to `GOVERNANCE_ROLE`.

5.4.14 Using `ERC1155Solmate` `MINTER_ROLE` to access control burn functions reduces separation of concerns

Severity: Low Risk

Context: ERC1155Solmate.sol#L28, ERC1155Solmate.sol#L91, ERC1155Solmate.sol#L99

Description: The `ERC1155Solmate` `MINTER_ROLE` is used to access control burn functions as well. This overlap reduces separation of concerns between the two distinct actions of minting and burning. It may add more clarity and better control to create a separate `BURNER_ROLE` which has access to the burn functions. This gives an option to use two different authorized addresses for minting and burning and thereby reduces the risk of a single compromised address misusing both mint and burn functions.

Recommendation: Consider adding a separate `BURNER_ROLE` which has access to the burn functions, and is distinct from the `MINTER_ROLE`.

Redacted: PR#17.

Spearbit: Reviewed that PR#17 fixes the issue as per the recommendation.

5.5 Gas Optimization

5.5.1 Event emission of `SetMaxBufferSize` can use function parameter to save gas

Severity: Gas Optimization

Context: PirexEthValidators.sol#L241-L243

Description: The event emission of `SetMaxBufferSize` can use the function parameter `_maxBufferSize` instead of the storage variable `maxBufferSize` to save gas by avoiding a SLOAD.

Recommendation: Consider changing event emission to `emit SetMaxBufferSize(_maxBufferSize);`

Redacted: PR#22.

Spearbit: Reviewed that PR#22 fixes the issue as per the recommendation.

5.5.2 Adding a `fromIndex != toIndex` check will save gas on inconsequential/incorrect validator queue swaps

Severity: Gas Optimization

Context: ValidatorQueue.sol#L70-L76

Description: `ValidatorQueue.swap` allows swapping the location of one validator with another. However, if the from and to indices are the same for inconsequential/incorrect swaps then this function can skip the swap logic to save gas.

Recommendation: Consider adding a `fromIndex != toIndex` check before performing the swap logic and return early otherwise.

Redacted: PR#15.

Spearbit: Reviewed that PR#15 fixes the issue as per the recommendation to revert with an error if the same index is used for the swap.

5.6 Informational

5.6.1 `PirexFees.distributeFees()` does not revert for non-existing fee tokens

Severity: Informational

Context: SafeTransferLib.sol#L9, PirexFees.sol#L6, PirexFees.sol#L84-L98

Description: As documented, Solmate's `SafeTransferLib` (unlike OpenZeppelin's `SafeERC20`) does not check whether the target address contains contract code. Use of this library's `safeTransferFrom()` in `PirexFees.distributeFees()` means that code will not revert if incorrectly used with a non-existing fee token. While the current implementation only uses this with `pxEth`, any reuse of this code in future revisions may be impacted.

Recommendation: Consider adding a `token.code.length > 0` check in `PirexFees.distributeFees()`.

Redacted: Acknowledged with no action taken at the moment considering the scope and usage of the `PirexFees` contract in the `PirexEth` system.

Spearbit: Acknowledged.

5.6.2 Misleading comment about validator status reduces readability

Severity: Informational

Context: DataTypes.sol#L39

Description: The comment: "*Withdrawable, // NO LONGER USED*" tends to convey that the validator status of `Withdrawable` indicates a validator or this enum itself not being used anymore. However, it is meant to indicate that the staking validator has initiated the withdrawal process upon redemption being triggered.

Recommendation: Update the comment to keep it in sync with implementation.

Redacted: PR#18.

Spearbit: Reviewed that PR#18 fixes the reported issue by updating the validator status explanations.

5.6.3 Enforcing the same `FEE_MAX` for all three fee types reduces flexibility

Severity: Informational

Context: PirexEth.sol#L19, PirexEth.sol#L83, DataTypes.sol#L21-L25

Description: The protocol has three fee types for deposits, redemptions and instant redemptions. However, it enforces the same `FEE_MAX` of 20% for all three fee types. It would provide more flexibility to have different maximum thresholds for the three types e.g. it could be reasonable for instant redemptions to have a fee > 20%.

Recommendation: Consider different max fees for the three different fee types.

Redacted: PR#6.

Spearbit: Reviewed that PR#6 fixes the issue by creating a `maxFees` mapping for each fee type.

5.6.4 The custom error used is different from the exception condition

Severity: Informational

Context: PirexEthValidators.sol#L237-L239

Description: The condition `_maxBufferSize < buffer` checked for `revert Errors.ExceedsMax()` is satisfied when resetting `maxBufferSize` incorrectly to a value <= current `buffer` value. So this is technically not exceeding max bounds but below min expected bounds.

Recommendation: Consider a new custom error such as `MinBoundExceeded()` for naming clarity.

Redacted: PR#22.

Spearbit: Reviewed that PR#22 fixes the issue as per the recommendation by adding a new custom error `InvalidMaxBuffer`.

5.6.5 Stale comment about `ERC1155Solmate.sol` roles reduces readability

Severity: Informational

Context: ERC1155Solmate.sol#L21-L23

Description: The comment: "*The account that deploys the contract will be granted the minter and pauser roles, as well as the default admin role, which will let it grant both minter and pauser roles to other accounts.*" does not match the implementation because there is no pauser role.

Recommendation: Update the comment or the implementation to keep them in sync.

Redacted: PR#16.

Spearbit: Reviewed that PR#16 fixes the reported issue by updating the comment.