Security Review Report
NM-0235 Monarch

NETHERMIND
SECURITY

(Jun 4, 2024)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the Monarch. It is a delegated staking protocol based on the **Pirex** that allows institutional investors to earn on their deposited Ether. Special security and compliance measures are taken to meet the institutional client needs. That involves restricting the deposit access only to the users that have verified their identity in the `Know Your Customer` (KYC) process. The earnings are gained from the Ethereum staking rewards, MEV, and block rewards. **Monarch** infrastructure performs all the necessary actions, like interaction with the `Beacon Chain` or running and maintaining the correct validation process. Monarch makes the whole process abstracted and convenient for the investor. The protocol consists of two main parts: `IntitutionalPirexEth` that maintains the staking process, including deposits, withdrawals, `Validators` maintenance, and the `AutoPxEth` Vault that accrues and distributes the rewards. The `Monarch` ensures investors' safety by employing a set of keepers that monitor the `Beacon Chain` state and act in critical scenarios such as slashing or inactivity leaks. Additionally, `Monarch` keeps part of the funds in the `Burner Accounts` that are used in those situations. The most important actions are performed by the `Governance`.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** eighteen points of attention, where three are classified as `Medium`, four are classified as `Low`, and eleven are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

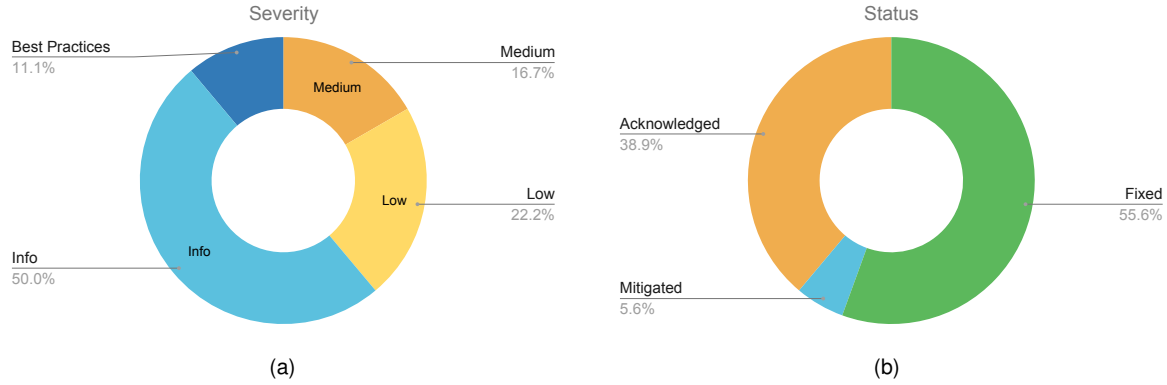

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (3), **Low** (4), **Undetermined** (0), **Informational** (9), **Best Practices** (2). **Distribution of status: Fixed** (10), **Acknowledged** (7), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	May 29, 2024
Response from Client	Regular responses during audit engagement
Final Report	Jun 4, 2024
Repository	dinero-pirex-eth
Commit (Audit)	4137f34d2b3d39ac5bf37e1e6bc87a7006f9a2b8
Commit (Final)	912ddb295dc36c31b87af0d02edaa9fb2bbd00fa
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	InstitutionalPirexEthValidators.sol	344	193	56.1%	45	582
2	InstitutionalPirexEth.sol	192	102	53.1%	16	310
3	RewardRecipientGateway.sol	131	118	90.1%	25	274
4	AutoPxEth.sol	206	198	96.1%	74	478
5	libraries/Errors.sol	45	127	282.2%	42	214
6	libraries/InstitutionalPirexEthValidatorManagementLogic.sol	40	27	67.5%	8	75
7	libraries/InstitutionalPirexEthWithdrawLogic.sol	361	180	49.9%	55	596
8	libraries/DataTypes.sol	107	131	122.4%	20	258
9	libraries/Constants.sol	9	34	377.8%	6	49
10	libraries/InstitutionalPirexEthConfigurationLogic.sol	248	166	66.9%	48	462
11	libraries/InstitutionalPirexEthValidationLogic.sol	108	98	90.7%	17	223
12	libraries/InstitutionalPirexEthGenericLogic.sol	14	15	107.1%	2	31
13	libraries/InstitutionalPirexEthDepositLogic.sol	283	143	50.5%	41	467
	Total	**2088**	**1532**	**73.4%**	**399**	**4019**

3 Summary of Issues

	Finding	Severity	Update
1	Emergency withdrawals are limited to buffer	Medium	Fixed
2	Platform fee paid in `pxETH` can't be exchanged for `ETH`	Medium	Fixed
3	Withdrawals can put the `InstitutionalPirexEth` contract in the incorrect state	Medium	Mitigated
4	Implementation contracts can be initialized	Low	Fixed
5	Investor's redemptions might cause Keeper's transactions to revert	Low	Fixed
6	The logic to charge the deposit fee is missing	Low	Acknowledged
7	Users are not able to withdraw under certain conditions	Low	Acknowledged
8	Burner accounts are not ensured to hold ipxETH	Info	Acknowledged
9	Cost of execution of the `deposit(...)` call is not constant	Info	Fixed
10	Implementation contracts contain real data	Info	Acknowledged
11	Lower boundary on the `maxBufferSizePct` is missing	Info	Acknowledged
12	Pre-deposit amount should be bounded	Info	Acknowledged
13	The Ether surplus from the max buffer decrease could be used for staking	Info	Fixed
14	The implementation contract `AutoPxEth` can't be used for both retail and institutional versions	Info	Acknowledged
15	`executeEmergencyWithdraw(...)` will revert due to overflow for some values	Info	Fixed
16	`iupxETH` holders may wait 36 days until withdrawal in case of slashing	Info	Fixed
17	Using `_____gap`	Best Practices	Fixed
18	Using `address(0)` for ETH	Best Practices	Fixed

4 System Overview

The **Monarch's** core functionality is defined in the contracts `InstitutionalPirexEth`, which is the main interface for the investors, and `InstitutionalPirexEthValidators`, which implements the logic related to operating the `Validators`. The `RewardRecipientGateway` contract receives the `Beacon Chain` staking rewards, from which they are distributed to other parts of the system. The `AutoPxEth Vault` handles all the reward calculation logic.

The diagram below showcases a high-level view of the system's architecture. The following sections delve into the system's components and their interactions.

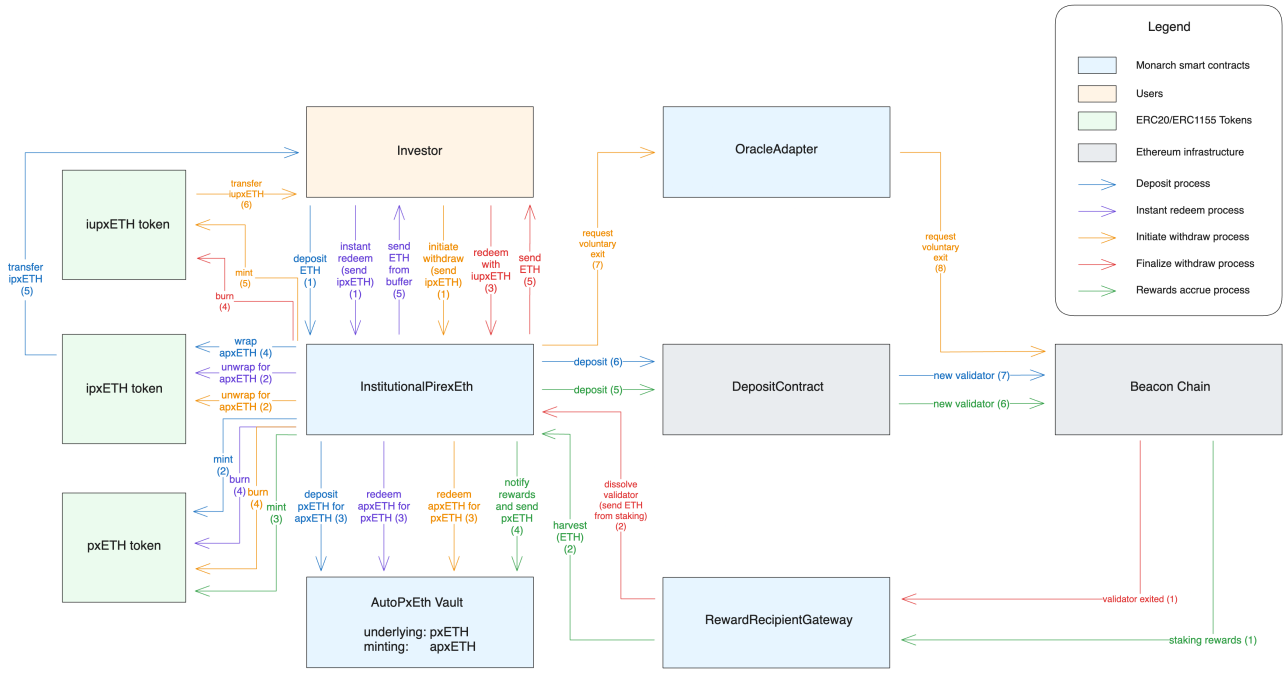

Fig. 2: Monarch - Structural Diagram of contracts. Note that the diagram does not present flows for slashing, top-ups, and emergency withdrawals.

4.1 Actors of the system

The following actors participate in the Monarch protocol:

- **Investors**: Investors are the protocol users. They are institutional clients that have verified their identity in the *Know Your Customer* (KYC) process. They are allowed to interact with the protocol by depositing funds.
- **Keeper**: Part of the off-chain infrastructure operated by the Monarch team. The keeper performs important protocol operations such as distributing the Ether from rewards or withdrawals, monitoring `Beacon Chain`, spinning up new `Validators` when necessary, and reacting to critical scenarios such as slashing or top-ups.
- **Governance**: Redacted Cartel's governance controls the protocol parameters, such as fee and buffer percentages. Governance can pause and unpause the protocol, add or remove the validators, and burner accounts that can be used in the system.
- **Oracle**: Part of the off-chain infrastructure that submits the *voluntary exit request* to the `Beacon Chain` to trigger the `Validator's` exit. It monitors the `Validator` exit process and triggers the *dissolve process*, which allows investors to finish the withdrawal process.
- **Burner accounts**: Protocol-controlled accounts; they are no different from regular users. They hold `ipxETH` that will be redeemed for `pxETH` and burned in case the buffer is used to cover a slash or a top-up.

4.2 Tokens used within the system

The following tokens are used within the Monarch protocol:

- **Native Ether**: Investors deposit native Ether into the `InstitutionalPirexEth` contract. A portion of Ether stays in the contract in the form of a security buffer, and the rest is sent to the Deposit Contract to register and activate new validators.

- **pxETH**: The underlying token of the `AutoPxEth` vault. For each Ether deposited, there is an equivalent amount of `pxETH` tokens minted and automatically deposited into the autocompounding vault. As opposed to the retail version of the system, in the institutional one, no one except for the vault should hold `pxETH` tokens. For the institutional version, a new separate deployment of `pxETH ERC20` will be made.
- **apxETH**: The `AutoPxEth` vault share token minted in return for `pxETH` deposits. Its value increases with the higher amount of `pxETH` tokens in the vault. Similarly to `pxETH`, `apxETH` is an intermediary token, and no user should hold it. It is always minted for the `InstitutionalPirexEth` contract and immediately wrapped for the `ipxETH`.
- **ipxETH**: The `apxETH` share tokens get wrapped into `ipxETH` and sent back to the investor. Because only the Monarch's smart contracts can wrap and unwrap `ipxETH`, no user can obtain `apxETH` and `pxETH`. This ensures that the institutional funds can only go through the flows defined in the code and are isolated from the retail users.
- **iupxETH**: The unlocked version of the `ipxETH` token that the investor receives after initiating the redemption process. It entitles its holder to withdraw Ether once the `Validator` finishes the exit procedure. The `iupxETH` token is an `ERC1155`, where each token ID is connected to a particular `Validator's` public key.

4.3 Deposits

The `deposit(...)` function of the `InstitutionalPirexEth` contract is the system's entry point for the users. Investors deposit native Ether into the protocol, and in return, they receive the `ipxETH` tokens. Based on the maximum buffer percentage set by the governance, part of the deposited Ether is set aside as a buffer. The keeper can utilize the buffer to fulfill his duties. Investors can use the buffer to instantly redeem their `ipxETH` tokens for Ether. The rest of the Ether is added to the `pendingDeposit`, ready to be sent to the `Deposit Contract`. Whenever `32 ETH` is gathered, a deposit will be made, and a new validator will enter the activation queue. Once the validator starts validating, all the staking rewards will go to the `RewardRecipientGateway` contract's address. From there, the keeper will distribute them as protocol rewards by calling the `harvest(...)` function.

4.4 Rewards

The `RewardRecipientGateway` contract receives the staking rewards from all validators operated by the protocol team - both retail and institutional. It is the only component shared between the two systems. Once the keeper calls `harvest(...)`, the rewards will be distributed among retail and institutions proportionally to the total assets staked in their respective vaults. Since the staking rewards are granted in the native Ether, once they are distributed, they will be reinvested into the protocol to launch new validators. An equivalent amount of `pxETH` will be minted and deposited into the `AutoPxEth` vault as a reward for existing investors.

4.5 Redemptions and withdrawals

Investors holding `ipxETH` tokens have two ways to exit the investment:

a. **Instant redemption**: Users can redeem their `ipxETH` tokens directly for Ether using the liquidity stored in the buffer. This option is faster since it does not require spinning down the validators and waiting for them to exit.

b. **Redemption and withdrawal**: The standard flow is a two step process. The investor first initiates the redemption with the `initiateRedemption(...)` function. This call triggers the oracle to send the voluntary exit request to the `Beacon Chain`. The investor receives the `ERC1155 iupxETH` tokens in return for his `ipxETH`. Each `token ID` is minted in the amount of `32`, where each `token ID` is assigned to a particular validator that entered the exit queue. Once the keeper confirms that Ether from staking returned to the `RewardRecipientGateway` contract, the investor can call `redeemWithIUpxEth(...)` to exchange the `iupxETH` tokens for Ether.

In both flows, the `ipxETH` tokens are first unwrapped into the vault share token `apxETH`. If there was a previous reward distribution, the shares would be worth more in terms of assets. During the redemption process, `apxETH` would be redeemed for more `pxETH`. Since the `pxETH` tokens represent Ether in a `1:1` ratio, the investor would earn Ether.

4.6 Validator slashing and top-ups

Validators get the staking rewards for attesting to blocks. Misbehavior, on the other hand, results in penalties. The penalties can be incurred due to a software bug or the off-chain infrastructure going offline. They don't necessarily have to be caused by malicious behavior. The keeper monitors the validators to ensure that the yield is generated in the protocol and that there are enough funds for every user to withdraw. If the validator's balance falls below `32 ETH` e.g., due to the inactivity leak, the keeper will call the `topUpStake(...)` function to add the missing Ether. In case of a slash, a validator that was previously staking will be forced to exit. In such a situation, the keeper would call the `slashValidator(...)` function that would mark that validator as slashed. After the staked `ETH` (minus penalty) is unlocked and sent to the recipient contract, the keeper would forward it to deploy a new validator. The amount of `ETH` must be `32`, so the missing part is taken from the treasury or the buffer. To compensate for the net outflow of Ether from the system, whenever the buffer is used to cover a slash or a top-up, an equivalent amount of `pxETH` is burned from the burner accounts. This ensures that the `1:1` ratio between `pxETH` and Ether is kept.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Medium] Emergency withdrawals are limited to buffer

File(s): InstitutionalPirexEthWithdrawLogic.sol

Description: The function `emergencyWithdraw(...)` allows to withdraw the funds in case of an emergency. The `InstitutionalPirexEth` contract can contain three types of ETH funds: pending deposits, redemptions and buffer. The function `emergencyWithdraw(...)` validates parameters, and then calls `executeEmergencyWithdraw(...)`, which is presented below:

```
function executeEmergencyWithdraw(
    DataTypes.PirexEthValidatorVars storage pirexEthValidatorVars, DataTypes.PirexEthValidatorContracts storage
    ↪  pirexEthValidatorContracts,
    uint256 amount, address token, address ipxEth, address receiver, mapping(address => bool) storage burnerAccounts,
    DataTypes.BurnerAccount[] memory $burnerAccounts ) external {
    InstitutionalPirexEthValidationLogic.validateEmergencyWithdraw(
        pirexEthValidatorVars.paused,
        amount,
        token,
        ipxEth,
        receiver
    );
    if (token == address(0)) {
        // Update pendingDeposit and outstandingRedemptions when
        // affected by emergency withdrawal
        uint256 remainingBalance = address(this).balance - amount - pirexEthValidatorVars.outstandingRedemptions;

        if (pirexEthValidatorVars.pendingDeposit > remainingBalance) {
            pirexEthValidatorVars.pendingDeposit = remainingBalance;
        }
        // burn pxEth
        InstitutionalPirexEthConfigurationLogic.updateBuffer( pirexEthValidatorVars, pirexEthValidatorContracts,
        ↪  burnerAccounts, amount, $burnerAccounts );

        // Handle ETH withdrawal
        (bool _success, ) = payable(receiver).call{value: amount}("");
        assert(_success);
    } else {
        ERC20_0(token).safeTransfer(receiver, amount);
    }
    emit EmergencyWithdrawal(receiver, token, amount);
}
```

At the end of a function, the amount of ETH specified in the `amount` is sent to the `receiver`. However, before the ETH transfer, the function `updateBuffer(...)` is called. It updates the buffer and burns the corresponding `ipxETH`. If the provided `amount` is less than the current buffer, the function reverts.

```
function updateBuffer(...) public {
    ////////////////////////////////////////////////////////////////////////
    // @audit: reverts if the available buffer is lower than requested amount
    ////////////////////////////////////////////////////////////////////////
    if (pirexEthValidatorVars.buffer < amount) {
        revert Errors.NotEnoughBuffer();
    }
    batchBurnPxEth( pirexEthValidatorVars, pirexEthValidatorContracts, burnerAccounts, amount, $burnerAccounts );
    pirexEthValidatorVars.buffer -= amount;
}
```

This effectively allows the function to burn buffer, but not to emergency withdraw the ETH in the contract. Note, that the buffer is planned to be around 5% of the whole deposited funds. This means, that if the contract holds funds as pending deposit and/or redemptions, and emergency withdrawal of ETH is needed, only funds in buffer can be withdrawn by burning, but no other funds.

Recommendation(s): Consider reviewing to define it's purpose. If it should only burn buffer, then the update of `pendingDeposit` is not needed, since it will never be reached. Alternatively, if the function should allow for withdrawal of ERC20 tokens and ETH, consider modifying the function so it allows for withdrawing funds outside the buffer.

Status: Fixed

Update from the client: Emergency withdraw performs withdrawal of funds in preference from buffer, pendingDeposit and outstandingRedemptions.

Update from Nethermind Security: The issue is fixed, as governance will be able to transfer all funds in case of emergency. However, in the current implementation, there is a redundant code in the `executeEmergencyWithdraw(...)`. During the `ETH` withdrawal, the last branch `else if (_remainingAmount > 0)` that updates `outstandingRedemptions` won't ever be executed because reaching this part of the function means that all funds are transferred out from the contract. But, the logic for transferring all funds is executed earlier in the branch `if (amount == _totalEthBalance)`.

6.2 [Medium] Platform fee paid in pxETH can't be exchanged for ETH

File(s): AutoPxEth.sol

Description: The `AutoPxEth` vault is a modified `ERC4626` with `pxETH` as the underlying token. The `pxETH` token is a one-to-one representation of the staked `ETH` and is a crucial component in the Monarch protocol. Deposited `pxETH` represents two forms of `ETH`: the staked `ETH`, and `ETH` from staking rewards. The rewards are taxed with the platform fee, which can be set up maximally to `20%`. This fee is deduced during a call to the `harvest(...)` function. The fee is paid with the `pxETH` token and transferred to the governance-controlled address.

```
function harvest() public updateReward(true) {
    uint256 _rewards = rewards;

    if (_rewards != 0) {
        rewards = 0;

        // Fee for platform
        uint256 feeAmount = (_rewards * platformFee) / FEE_DENOMINATOR;

        // Deduct fee from reward balance
        _rewards -= feeAmount;

        // Claimed rewards should be in pxETH
        ERC20(asset()).safeTransfer(platform, feeAmount);

        // Stake rewards sans fee
        _stake(_rewards);

        emit Harvest(msg.sender, _rewards);
    }
}
```

However, this transferred `pxETH` can't be exchanged for `ETH` and therefore is worthless. This is because the Monarch allows for exchanging `ETH` to `ipxETH` and `ipxETH` to `ETH`. The `pxETH` and `apxETH` are intermediary tokens and should never be outside of the `AutoPxEth` or `InstitutionalPirexEth` contracts. Consider the token paths for deposit and withdrawal:

- deposit: `ETH` -> `pxETH` -> `apxETH` -> `ipxETH`;
- withdraw (instant): `ipxETH` -> `apxETH` -> `pxETH` -> `ETH`;
- withdraw (two-step): `ipxETH` -> `iupxETH` -> `apxETH` -> `pxETH` -> `ETH`;

When the `pxETH` is transferred to the governance-controlled address, it can't be exchanged for `ETH`, since the only way to get `ETH` is to redeem the `ipxETH` through the `InstitutionalPirexEth`, and the only way to get `ipxETH` is by providing `ETH` to the `InstitutionalPirexEth` as well. Note that `pxETH` for institutions is a different token than `pxETH` for retail. The only thing that can be done with this `pxETH` is to transfer it back to the `AutoPxEth` vault, which would increase the value of `ipxETH`. With the current setup, the platform fee freezes funds and can't be effectively deducted from rewards for the governance.

Recommendation(s): Consider removing the platform fee or charging it in a way that allows exchanging it for `ETH`. One possible approach could be implementing the function in `InstitutionalPirexEth` that would allow sending the fee amount of `pxETH` to burn it for an equivalent amount of `ETH` from the buffer.

Status: Fixed

Update from the client: We have created a separate vault to receive platform fee in pxEth which can allow redemption directly into ETH to iupxEth. The reason for the same is not held pxEth outside of institutional PirexEth system.

Update from Nethermind Security: Since a new contract and role have been introduced with this fix, consider reviewing the deployment scripts to make sure that the new contract `InstitutionalPlatformFeeReceiver` is correctly granted required roles (e.g., `minter` and `burner` roles for `ipxEth` token) and that the new role `INSTITUTIONAL_PLATFORM_ROLE` is correctly granted.

6.3 [Medium] Withdrawals can put the `InstitutionalPirexEth` contract in the incorrect state

File(s): `InstitutionalPirexEth.sol`

Description: Withdrawal call to the `InstitutionalPirexEth` may result in an incorrect voluntary exit request to the beacon chain, which would lead to minting `iupxETH`, removing the validator from the `stakingValidators` queue, and setting the status of the validator as `Withdrawable`, while the validator does not enter exit queue on the beacon chain. This will happen if the withdrawal is made within 28 hours from the deposit of the full amount of `ETH`. `InstitutionalPirexEth` marks the validator with the status `Staking` right after the full amount is collected in the `pendingDeposits` and sent to the `DepositContract`, as presented in the function below:

```
function deposit(...) public {
  uint256 remainingCount = pirexEthValidatorVars
      .maxProcessedValidatorCount;
  uint256 _remainingdepositAmount = depositSize - preDepositAmount;

  while (
      initializedValidators.count() != 0 &&
      pirexEthValidatorVars.pendingDeposit >= _remainingdepositAmount &&
      remainingCount > 0
  ) {
      // ...

      (bool success, ) = beaconChainDepositContract.call{
          value: _remainingdepositAmount
      }(
          abi.encodeCall(
              IDepositContract.deposit,
              (
                  validatorParams.pubKey,
                  validatorParams.withdrawalCredentials,
                  validatorParams.signature,
                  validatorParams.depositDataRoot
              )
          )
      );

      // ...
      // @audit-note: Validator is marked as Staking right after the deposit of the full amount
      status[validatorParams.pubKey] = DataTypes.ValidatorStatus.Staking;

      stakingValidators.add(
          DataTypes.Validator(
              validatorParams.pubKey,
              validatorParams.signature,
              validatorParams.depositDataRoot,
              validatorParams.receiver
          ),
          validatorParams.withdrawalCredentials
      );

      emit ValidatorDeposit(validatorParams.pubKey);
  }
}
```

However, from the moment of depositing the `32 ETH` to the beacon chain, there are two stages in which the voluntary exit request of the validator can't be processed. The first stage is an activation queue which lasts for about `26` minutes (`MAX_SEED_LOOKAHEAD`). After that, the validator is activated, but to be allowed for the voluntarily exit it must be active for `27` hours (`SHARD_COMMITTEE_PERIOD`). The `process_voluntary_exit(...)` function from the beacon chain specification:

```python
def process_voluntary_exit(state: BeaconState, signed_voluntary_exit: SignedVoluntaryExit) -> None:
    voluntary_exit = signed_voluntary_exit.message
    validator = state.validators[voluntary_exit.validator_index]
    # @audit-note: Validator must be active (exit activation queue) to request voluntary exit
    assert is_active_validator(validator, get_current_epoch(state))
    assert validator.exit_epoch == FAR_FUTURE_EPOCH
    assert get_current_epoch(state) >= voluntary_exit.epoch
    # @audit-note: Validator must be active for 27 hours to request voluntary exit
    assert get_current_epoch(state) >= validator.activation_epoch + SHARD_COMMITTEE_PERIOD
    domain = get_domain(state, DOMAIN_VOLUNTARY_EXIT, voluntary_exit.epoch)
    signing_root = compute_signing_root(voluntary_exit, domain)
    assert bls.Verify(validator.pubkey, signing_root, signed_voluntary_exit.signature)
    initiate_validator_exit(state, voluntary_exit.validator_index)
```

As can be seen, a voluntary exit request won't be processed if made under `28 hours` from depositing. But the `initiateRedemption(...)` in `InstitutionalPirexEth` assumes that the request is always successful:

```solidity
function initiateRedemption(...) public {
    pirexEthValidatorVars.pendingWithdrawal += pxEthAmount;

    while (pirexEthValidatorVars.pendingWithdrawal / depositSize != 0) {
        // ...
        // @audit-note: Exit request through the oracle assumes always success
        pirexEthValidatorContracts.oracleAdapter.requestVoluntaryExit(
            _pubKey
        );

        batchIdToValidator[pirexEthValidatorVars.batchId++] = _pubKey;
        status[_pubKey] = DataTypes.ValidatorStatus.Withdrawable;
    }
}
```

The consequence of this assumption is that the `initiateRedemption(...)` function can change the validator state to `Withdrawable`, remove the validator from the `stakingValidators` queue, and mint the `iupxETH`, but leave the validator status as active on the beacon chain. This would leave `iupxETH` holders expecting that the validator is in the exit queue, which, in reality, is not. Additionally, depending on how the Monarch's off-chain infrastructure is set up, the exit request from the oracle may cause Monarch to stop validating the transactions, leading to an inactivity leak and loss of funds. However, if the Monarch team had noticed that situation and correctly sent the exit request to the beacon chain, the issue would have been resolved.

Recommendation(s): Consider allowing to initiate a withdrawal for a given validator only after `28` hours (or more) from the deposit of the full amount to the `DepositContract`.

Status: Mitigated

Update from the client: Decided to keep as is. The off-chain keeper job will handle the voluntary exit of the validator yet to be in active_ongoing state.

6.4 [Low] Implementation contracts can be initialized

File(s): `InstitutionalPirexEth.sol`, `AutoPxEth.sol`, `RewardRecipientGateway.sol`

Description: The contracts `InstitutionalPirexEth`, `AutoPxEth`, and `RewardRecipientGateway` do not disable the `initialize(...)` functions. This allows anyone to call `initialize(...)` with arbitrary values, which may lead to unwanted behavior. One example of such malicious behavior is initiating a call from the implementation contracts to sanctioned or black-listed contracts, which may lead to legal issues for the Monarch protocol, which could be especially dangerous for institutions using the project.

Recommendation(s): Consider removing the possibility of calling the initializers directly on the implementation contracts. This may be done by calling the `_disableInitializers(...)` in the constructor, which will disable calling functions with the `initializer` modifier.

Status: Fixed

Update from the client: Added a call to `_disableInitializers` to the constructor of implementation contracts viz `WrappedToken`, `InstitutionalPirexEth`, `RewardRecipientGateway` and `AutoPxEth`.

6.5 [Low] Investor's redemptions might cause Keeper's transactions to revert

File(s): `InstitutionalPirexEthConfigurationLogic.sol`

Description: Some operations performed by the Keeper, such as topping up and slashing the validator, are allowed to use the `ETH` buffer. The `ETH` amount taken from the buffer must be compensated by burning an equivalent amount of `pxETH` from the burner accounts. Whenever buffer is used, both `executeTopUpStake(...)` and `slashValidator(...)` functions call the `updateBuffer(...)` function from the `InstitutionalPirexEthConfigurationLogic` library to first check if there is enough Ether in the buffer to fulfill the operation and later to burn the `pxETH` tokens.

```
function updateBuffer(
    DataTypes.PirexEthValidatorVars storage pirexEthValidatorVars, DataTypes.PirexEthValidatorContracts storage
    ↪  pirexEthValidatorContracts,
    mapping(address => bool) storage burnerAccounts, uint256 amount, DataTypes.BurnerAccount[] memory $burnerAccounts
) public {
    // @audit We want to use `amount` of ETH from the buffer.
    if (pirexEthValidatorVars.buffer < amount) {
        revert Errors.NotEnoughBuffer();
    }
    // @audit Burn `amount` of pxETH from burner accounts
    // to compensate for the outflow of ETH.
    batchBurnPxEth( ..., burnerAccounts, amount, $burnerAccounts );
    // @audit Use the buffer
    pirexEthValidatorVars.buffer -= amount;
}
```

The `batchBurnPxEth(...)` function transfers the `ipxETH` tokens from the burner accounts, unwraps them into the `apxETH` share tokens, and then redeems the shares in the `AutoPxEth` vault for `pxETH` tokens to be burned.

```
function batchBurnPxEth( ..., mapping(address => bool) storage burnerAccounts, uint256 amount,
    DataTypes.BurnerAccount[] memory $burnerAccounts ) public {
    uint256 _len = $burnerAccounts.length;
    uint256 _sum;
    ...
    for (uint256 _i; _i < _len;) {
        // ...
        // @audit Transfer ipxETH from burner account to this contract
        _ipxEthERC20.safeTransferFrom(
            $burnerAccounts[_i].account, address(this), $burnerAccounts[_i].amount
        );
        // @audit Unwrap ipxETH to apxETH
        pirexEthValidatorContracts.institutionalPxEth.unwrap(
            address(this), $burnerAccounts[_i].amount
        );
        // @audit Redeem apxETH for pxETH
        uint256 _pxEthAmount = pirexEthValidatorContracts.autoPxEth.redeem(
            $burnerAccounts[_i].amount, address(this), address(this)
        );
        // @audit Burn pxETH
        burnPxEth( ..., _pxEthAmount );
        // @audit The sum gets increased by the amount redeemed from the vault.
        _sum += _pxEthAmount;
        unchecked {
            ++_i;
        }
    }
    // @audit Strict equality check might revert when the sum of the actual
    // redeemed pxETH was different than initially expected.
    assert(_sum == amount);
}
```

The problem present in the `batchBurnPxEth(...)` is that, in some cases, the transaction might revert due to a slight change in the assets to shares ratio in the vault, which will impact the amount of `pxETH` redeemed. Consider the following scenario:

- Keeper calls the `slashValidator(...)` function. He wants to use the buffer to cover the slash. He computed the `amount` of Ether that needs to be taken from the buffer off-chain and expects that the `sum` of `ipxETH` unwrapped, redeemed, and burned from burner accounts to cover that Ether `amount`;
- At the same time, the Investor initiates a redemption process. His `ipxETH` gets unwrapped to `apxETH`, which gets redeemed in the vault for `pxETH` minus the withdrawal penalty that stays in the vault;

If the Investor specifies a higher gas price for his transaction than the Keeper, the redemption TX will be included in the block first. The withdrawal penalty paid by the Investor in `pxETH` works like a reward for other vault shareholders. When redeeming, they will get more `pxETH` for the same amount of shares. This causes an issue when the Keeper's transaction gets executed. Because there is more `pxETH` in the vault, the total sum of assets redeemed on behalf of the burner accounts will be higher than what Keeper initially calculated. The strict equality check at the end of the `batchBurnPxEth(...)` function will revert, and the gas spent by the Keeper will be wasted.

As a result, during periods of high network activity, when gas prices dynamically change, top-ups and slashing operations performed by the Keeper might often revert.

Recommendation(s): To keep the `pxETH` amount aligned with Keeper's assumptions, the call to `redeem(...)` could be replaced with a call to `withdraw(...)`. The former accepts the shares to redeem and the latter the assets to receive. The exact amount of `ipxETH` to transfer

from the burner accounts could be computed with the help of the `previewWithdraw(...)` function. It would ensure that after the `ipxETH` is unwrapped into `apxETH` and withdrawn from the Vault, there are no leftover `apxETH` tokens in the contract.

Batching similar operations together could simplify the logic inside the `batchBurnPxEth(...)` function. For example, first transfer `ipxETH` from all the burner accounts and then perform the unwrapping, withdrawing, and burning with all the tokens once.

Status: Fixed

Update from the client: Refactored `InstitutionalPirexEthConfigurationLogic.batchBurnPxEth(...)` as per recommendation.

Update from Nethermind Security: The issue is not fixed. Introduced changes prevent unexpected revert. However, the problem of difference of `ipxETH` worth is still present and may lead to loss of funds. The `batchBurnPxEth(...)` function takes two amounts as an input: amount and `$burnerAccounts.amount`, where the first one represents the desired amount of `pxETH`, and the second the amount of `ipxETH` that will be transferred from burner accounts. The amount of `ipxETH` is computed off-chain, so the scenario described in the issue above, where the `ipxETH` is worth more due to realizing rewards, is still possible. In such case, if the provided amount of `ipxETH` is worth more than the provided amount of `pxETH`, the `ipxETH` will be unwrapped to `apxETH`, but then not all `apxETH` will be used to redeem in a vault since the specified amount of `pxETH` is lower than the worth of given `apxETH`. As a result, the additional `apxETH` stays in the `Pirex` contract and is not utilized. To fix this problem, consider specifying only the desired `pxETH` amount as the input parameter. Then, using `previewWithdraw(...)`, compute the amount of `ipxETH` that needs to be transferred from burner accounts. Next, transfer `ipxETH` looping through burner accounts, checking their balances, and updating the remaining amount. Unwrap collected `ipxETH`, redeem, and burn `pxETH`.

Update from Nethermind Security: The issue has been fixed.

6.6 [Low] The logic to charge the deposit fee is missing

File(s): InstitutionalPirexEthDepositLogic.sol

Description: The `InstitutionalPirexEth` contract uses a set of fees that users will be charged when performing certain operations. The possible fee types declared in the `Fees` enum in the `DataTypes` library are: `Deposit`, `Redemption`, and `InstantRedemption`. The fee amounts are controlled with the `setFee(...)` and `setMaxFee(...)` functions. The problem present in the `InstitutionalPirexEth` contract is that the `Deposit` fee is not being charged on the deposit operations. Given the fact that the max value for the `Deposit` fee is being set in the `initialize(...)` function of the `InstitutionalPirexEth` contract, we assume that the value of this fee might change over time when the governance decides to change it with the `setFee(...)` function.

```
function initialize(...) public override initializer {
    // ...
    // @audit The max deposit fee is set but
    // the fee is not charged in the protocol
    maxFees[DataTypes.Fees.Deposit] = 200_000;
    maxFees[DataTypes.Fees.Redemption] = 200_000;
    maxFees[DataTypes.Fees.InstantRedemption] = 200_000;
    // ...
}
```

The effect of such change will have no effect since the logic to charge the deposit fee is missing in the `deposit(...)` flow.

```
// @audit assets is msg.value sent by the Investor.
function executeDeposit(..., uint256 assets, ...)
    external returns (uint256 amount) {
    // ...

    // @audit Mint assets amount of pxETH to address(this).
    InstitutionalPirexEthConfigurationLogic.mintPxEth(...);

    // @audit Deposit all pxETH into the vault and receive apxETH vault shares.
    uint256 apxEthShares =
        pirexEthValidatorContracts.autoPxEth.deposit(...);

    // @audit Wrap vault shares into ipxETH and transfer to the receiver.
    // The logic for taking a deposit fee in ipxETH is missing.
    // The receiver will receive everything.
    pirexEthValidatorContracts.institutionalPxEth.wrap(
        receiver, apxEthShares
    );

    // @audit Proceed with the deposit.
    addPendingDeposit(...);

    // @audit The investor did not pay the deposit fee.
    emit Deposit(msg.sender, receiver, assets, apxEthShares);

    // ...
}
```

Recommendation(s): Even if it is a conscious design decision not to charge the institutional clients with a deposit fee, consider either adding logic to charge the fee and set its value explicitly to `0` or removing the redundant logic regarding the `Deposit` fee.

Status: Acknowledged

Update from the client: Removed redundant logic regarding the `Deposit` fee

Update from Nethermind Security: The `maxFees[DataTypes.Fees.Deposit]` was removed from the `initialize(...)` function, but the `Deposit fee` type is still present in the `Fees` enum. It is still possible to set the `Deposit` fee even though it is not being used.

6.7 [Low] Users are not able to withdraw under certain conditions

File(s): `InstitutionalPirexEth.sol`

Description: Users that deposit `ETH` in the `InstitutionalPirexEth` contract in exchange for the `ipxETH` token should be able to exchange it back to the `ETH`. This exchange can be instant or with a delay that requires holding the `iupxETH` token while the validator is waiting in the exit queue. However, under some circumstances, users won't be able to withdraw `ETH` for the `ipxETH`. Below, we list those situations:

- The contract may pause deposits by setting `depositEtherPaused == PAUSED`. In such a state, the users may deposit `ETH` to receive `ipxETH`, but new validators are not activated. If there are no currently running validators, users can deposit `ETH`, but the amount of withdrawn `ETH` is only up to a buffer size. The withdrawal of the rest of `ETH` requires spinning down the validators, which were not activated in the first place, and therefore, the `ETH` can't be withdrawn;
- There is an edge case where the `initializedValidators` queue is empty, and only one active validator gets slashed. In this case, after the keeper calls `slashValidator(...)`, the amount of `ETH` is kept in the `pendingDeposit`, but the new validator is not activated because there are no validators in the `initializedValidators` queue. This leads to a state where investors hold the `ipxETH` token, but it can't be exchanged for `ETH` (except the buffer amount);

Recommendation(s): Consider documenting cases where the user cannot withdraw `ETH`. Alternatively, consider allowing investors to withdraw `ETH` in presented cases. This can be done by providing changes, which may include:

- Allowing investors to instantly redeem `ipxETH` using the `ETH` registered in `pendingDeposit` during the deposit pause;
- In the described slashing scenario, allowing investors to redeem `ipxETH` without registering and activating a new validator, if there are no validators in the `initializedValidators` queue;

Status: Acknowledged

Update from the client: The theme Ethereum proof of stake is mirrored in Monarch's institutional liquid staking protocol, which means the user deposits must face the validator life cycle. Moreover, the depositors who actually deposited during the state where the initialized validator queue is empty might not be able to take advantage of withdrawing ETH from `pendingDeposit` due to race conditions. We can have the governance role to pause user deposits by setting `pirexEthValidatorVars.paused = PAUSED` if planning to discontinue this protocol or in case of initialized validator staking queue to be empty.

6.8 [Info] Burner accounts are not ensured to hold ipxETH

File(s): `InstitutionalPirexEth.sol`

Description: The burner accounts are special addresses that hold `ipxETH` tokens, which can be burned when using buffer funds. However, the contract does not ensure that those accounts hold any `ipxETH` tokens, increasing the size of a buffer.

Recommendation(s): Consider ensuring that burner accounts hold enough `ipxETH` during buffer updates.

Status: Acknowledged

Update from the client: Decided to keep it as is. The burner accounts holding `ipxEth` are for compensating any ETH used from buffer during `topUpStake(...)` or `slashValidator(...)`. However, the governance might also supply the required ETH from, say, multi-sig treasury, thereby not using from the buffer.

6.9 [Info] Cost of execution of the `deposit(...)` call is not constant

File(s): `InstitutionalPirexEth.sol`

Description: The investors calling the `deposit(...)` function would pay different amounts of gas depending on the collected pending deposit. If the value in `pendingDeposit` is high enough, after minting `ipxETH` to the investor, the `while` loop in the `InstitutionalPirexEth-DepositLogic.deposit(...)` function will be executed. This would involve costly actions such as modifying the contract's storage, calling the `DepositContract`, and minting an additional amount of `ipxETH` for pre-deposit. This disincentivizes the investors to make large deposits, which would result in executing the actions in the `while` loop once or multiple times. Instead, the small deposits that won't trigger additional actions are incentivized.

Recommendation(s): Consider modifying deposit logic, so that it does not penalise large deposits. This can be done by introducing optional validator activation during deposit. Alternatively, the activation of validators may be done exclusively by keepers.

Status: Fixed

Update from the client: Added an argument `_allowBeaconDeposit` to `deposit(...)` where deposit can opt-in to stake validator.

6.10 [Info] Implementation contracts contain real data

File(s): `InstitutionalPirexEth.sol`, `InstitutionalPirexEthValidators.sol`, `RewardRecipientGateway.sol`

Description: At the construction time, the implementation contracts set immutable variables, which include addresses of crucial contracts for the Monarch protocol. However, storing important data in the implementation contract cause a confusion of users and integrators and lead to using implementation contract instead of proxy.

Recommendation(s): Consider leaving the data in the implementation contract empty and setting them only for the proxy contract.

Status: Acknowledged

Update from the client: Decided to keep as is.

6.11 [Info] Lower boundary on the `maxBufferSizePct` is missing

File(s): `InstitutionalPirexEthConfigurationLogic.sol`

Description: The Monarch protocol employs the buffer, a part of Ether that is set aside and not used in the staking process. The buffer provides liquidity for instant redeems but can also be used by the Keeper to compensate for the Ether lost due to slashing or inactivity leaks. The `setMaxBufferSizePct(...)` function is used to control the buffer size as a percentage of total Ether deposited into the protocol. The problem with this function is that it does not constrain the lower boundary of the new buffer percentage. Therefore, it is possible for the Governance to set the buffer to zero percent either by accident or due to a governance attack. In such a scenario, there would be no instant liquidity available for institutional clients to exit the investment without waiting for the spin-down of the validators. With no buffer, it wouldn't also be possible to use burner accounts for critical operations such as slashing and top-ups.

Recommendation(s): Consider setting a reasonable lower bound for the new maximum buffer size percentage to eliminate the above-mentioned risks.

Status: Acknowledged

Update from the client: Decided to keep it as is. `topUpStake(...)` and `slashValidator(...)` do have an option where it can be supplied with externnal ETH.

6.12 [Info] Pre-deposit amount should be bounded

File(s): `InstitutionalPirexEth.sol`

Description: The pre-deposit amount is not bound to any maximum value. However, if the pre-deposit is high, due to the hysteresis added to the calculation of the effective validator balance, there is an edge case which can keep validator inactive even though the full amount of `32 ETH` was sent. In effect, if the pre-deposit would be `31 ETH` and the remaining `1 ETH` would be provided through the Monarch contracts, the validator wouldn't be considered active, even with the provided full `32 ETH`, and would require additional funds to activate.

Recommendation(s): Consider limiting the amount of pre-deposit to avoid the presented scenario.

Status: Acknowledged

Update from the client: Decided to keep as is. The workaround will be to call `topUp(...)` if such an edge case happens.

6.13 [Info] The Ether surplus from the max buffer decrease could be used for staking

File(s): `InstitutionalPirexEthValidators.sol`

Description: The Governance uses the `setMaxBufferSizePct(...)` function to set the maximum percentage the buffer can constitute of the whole Ether deposit. The acceptable values range from `0` to `100` percent. Only the `maxBufferSizePct` and `maxBufferSize` change when updating the maximum buffer. The current `buffer` remains the same.

```
function executeSetMaxBufferSizePct(
    DataTypes.PirexEthValidatorVars storage pirexEthValidatorVars,
    DataTypes.PirexEthValidatorContracts storage pirexEthValidatorContracts,
    uint256 pct
) external {
    InstitutionalPirexEthValidationLogic.validateSetMaxBufferSizePct(pct);
    // ...
    // @audit Max buffer percentage and value get updated, but the current buffer is not.
    pirexEthValidatorVars.maxBufferSizePct = pct;
    pirexEthValidatorVars.maxBufferSize = (
        pirexEthValidatorContracts.pxEth.totalSupply() * pct
    ) / Constants.DENOMINATOR;
}
```

When the new max buffer percentage is set to a lower value than before, the max buffer will be updated, but the buffer itself will remain above the new maximum. This situation is not problematic since the buffer can still be utilized as usual. However, the excess of the current buffer over the new maximum could be used to spin up new validators and generate a higher yield for the protocol's users. Not utilizing this surplus of Ether means that possible earnings are lost.

Recommendation(s): The described issue poses no security risk, but the current behavior might result in an unexpected yield loss. Therefore, the client may want to use that Ether to increase the amount of staking funds. However, note that such an operation would decrease the buffer that acts as a security fund for important cases.

Status: Fixed

Update from the client: We have refactored `InstitutionalPirexEth.setMaxBufferSizePct(...)` so that any excess buffer upon decrease in `maxBufferSize` can be transferred to `pendingDeposit` and hence used for staking new validators. Moreover, the update to `pendingDeposit`, `buffer` and `maxBufferSize` is made system wide upon change in pxEth totalSupply (`mint` and `burn`).

6.14 [Info] The implementation contract `AutoPxEth` can't be used for both retail and institutional versions

File(s): `AutoPxEth.sol`

Description: The `AutoPxEth` contract is an implementation contract that will be used in a proxy. During a contract construction time, the immutable variable `redirectEnabled` is set. It specifies if, during the transfer of the `apxETH` tokens (vault shares), the `PirexEth` contract should be called.

```
function transfer(
    address to,
    uint256 amount
) public override(ERC20Upgradeable, IERC20) returns (bool) {
    super.transfer(to, amount);

    if (redirectEnabled && to == address(pirexEth)) {
        pirexEth.initiateRedemption(amount, msg.sender, false);
    }

    return true;
}
```

Such design suggests, that this contract will be used as the implementation for the retail and institution vaults. However, because the immutable variable `redirectEnabled` is set at the construction time, this value can't be set up differently for different proxies. This leads to the conclusion that the `AutoPxEth` contract can't be used for both retail and institutional vault proxies, and two separate implementations are needed. Note that in this case, the variable `redirectEnabled` is unnecessary for both versions, as well as the modifications to the `transfer(...)` and `transferFrom(...)` for the institutional version.

Recommendation(s): Consider setting the value `redirectEnabled` during the initialization of the proxy contract to reuse the `AutoPxEth` implementation contract. Alternatively, separate implementation contracts for retail and institutions. The latter can be created without the redundant code, as proposed above.

Status: Acknowledged

Update from the client: Different AutoPxEth is being used for Retail and Institution already.

6.15 [Info] `executeEmergencyWithdraw(...)` will revert due to overflow for some values

File(s): `InstitutionalPirexEth.sol`

Description: The `emergencyWithdrawal(...)` does the following subtraction:

```
uint256 remainingBalance = address(this).balance -
            amount -
            pirexEthValidatorVars.outstandingRedemptions;
```

This may result in overflow since the input parameter `amount` may be higher than `address(this).balance` minus `pirexEthValidatorVars.outstandingRedemptions`. The resulting error won't be informative which can be especially improtant in case of emergency.

Recommendation(s): Consider ensuring that the input parameter `amount` is higher or equal to `address(this).balance - pirexEthValidatorVars.outstandingRedemptions` to mitigate the possible overflow.

Status: Fixed

Update from the client: This would become irrelevant as now emergency withdraw would withdraw all of the ETH held in `InstitutionalPirexEth` contract.

6.16 [Info] `iupxETH` holders may wait 36 days until withdrawal in case of slashing

File(s): `InstitutionalPirexEth.sol`

Description: The investors can call `initiateRedemption(...)` to initiate the process of the validator voluntary exit and receive `iupxETH` tokens. The investors may expect the validator to exit after about `28` hours, and their `iupxETH` tokens will be redeemable for Ether. However, while the validator waits for the exit, it can still be slashed, and the waiting period would change from `28` hours to `36` days since this is the time that the validator waits after it gets slashed. This significant time difference can affect institutional investors' processes and procedures. This particular scenario is not currently described in the documentation for the Pirex retail, which may leave some users unaware of this possible scenario.

Recommendation(s): Consider clearly documenting that such a scenario is possible and the predicted behavior of the Monarch platform to ensure that institutional investors are aware of and understand that mechanism.

Status: Fixed

Update from the client: Added documentation to `bulkRedeemWithIUpxEth(...)` as well `redeemWithIUpxEth(...)` for the delay in `iupxEth` during validator slashing while exiting

6.17 [Best Practice] Using `_____gap`

File(s): `AutoPxEthStorage.sol`, `InstitutionPirexEthStorage.sol`

Description: The contracts `AutoPxEthStorage` and `InstitutionPirexEthStorage` introduce `_____gap`, which allows for allocating storage space that can be utilized with new variables during upgrades of inherited contracts without shifting down the storage. However, below we list some best practices that can be applied for the `_____gap`:

- the size of a gap should sum with the number of already allocated storage slots to a round number (e.g., `50`). Consider this OZ ERC20 contract;
- gap can be used as the first variable in a contract to allow for the safe inheriting of new contracts;

Recommendation(s): Consider applying listed best practices for the `_____gap`.

Status: Fixed

Update from the client: Added `_____gap` as the first variable of the storage layout.

Update from Nethermind Security: The issue is fixed. To make future upgrades easier to perform, consider keeping the number of taken storage slots to an even number. For example, contract A defines three state variables, so the `_____gap` could use `47` slots (`_____gap[47]`). This way, the count of all state variables sums up to an even number of `50`. Future addition of `1` new state variable would simply require decreasing the gap slots by one from `47` to `46` (`_____gap[46]`).

6.18 [Best Practice] Using `address(0)` for ETH

File(s): `InstitutionalPirexEthWithdrawLogic.sol`

Description: In the `executeEmergencyWithdraw(...)` function the ETH is defined as `address(0)`. This may lead to accidental pointing to ETH where, in fact, the token address parameter was missed during the function call.

Recommendation(s): Consider using the non-zero special address to identify ETH. A popular choice is `address(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE)`.

Status: Fixed

Update from the client: Using `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` to represent ETH.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the Monarch documentation

The documentation for the Monarch is contained on the Dinero website. It refers to the retail version, which is similar to the audited project in general.
The team answered every question during meetings or through messages, which gave the auditing team a lot of insight and a deep understanding of the technical aspects of the project
Code comments are also of high quality, with explanations for every function describing the purpose, context, and situations where the function will be called. Other than functions, some comments exist in other areas of the code where extra information is necessary, allowing readers to understand the codebase at a faster pace.

8 Test Suite Evaluation

```
forge test
[] Compiling...
[] Compiling 155 files with 0.8.25
[] Solc 0.8.25 finished in 32.97s
Compiler run successful with warnings:
Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.25;"
--> test/ZapCurveIpxEthStEth.t.sol

Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.25;"
--> test/ZapEigenLayerIpxEth.t.sol

Warning (2519): This declaration shadows an existing declaration.
   --> src/scripts/InstitutionalDeploy.s.sol:292:9:
    |
292 |         address _institutionalPirexEth,
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> src/scripts/InstitutionalDeploy.s.sol:40:5:
   |
40 |     InstitutionalPirexEth _institutionalPirexEth;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> src/scripts/InstitutionalDeploy.s.sol:294:9:
    |
294 |         address _institutionalApxEth,
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> src/scripts/InstitutionalDeploy.s.sol:38:5:
   |
38 |     AutoPxEth _institutionalApxEth;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> src/scripts/InstitutionalDeploy.s.sol:347:9:
    |
347 |         address _institutionalApxEth
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> src/scripts/InstitutionalDeploy.s.sol:38:5:
   |
38 |     AutoPxEth _institutionalApxEth;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2018): Function state mutability can be restricted to view
   --> test/AutoPxEth.t.sol:288:5:
    |
288 |     function testAssetsPerShare() external {
    |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
  --> test/UpxEth.t.sol:57:5:
   |
57 |     function testSupportInterface() external {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Ran 3 tests for test/UpxEth.t.sol:UpxEthTest
[PASS] testBurnBatch() (gas: 90406)
[PASS] testMintBatch() (gas: 73659)
[PASS] testSupportInterface() (gas: 9356)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 13.91ms (1.04ms CPU time)
```

```
Ran 7 tests for test/OracleAdapter.t.sol:OracleAdapterTest
[PASS] testCannotDissolveValidatorUnauthorised() (gas: 12437)
[PASS] testCannotSetContractUnauthorized() (gas: 15654)
[PASS] testCannotSetContractUnrecognised() (gas: 10456)
[PASS] testCannotSetContractZeroAddress() (gas: 8863)
[PASS] testRequestVoluntaryExitUnauthorised() (gas: 11492)
[PASS] testSetContract() (gas: 23811)
[PASS] testVountaryExit(bytes) (runs: 101, : 20884, ~: 20558)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 28.80ms (15.64ms CPU time)

Ran 9 tests for test/PxEth.t.sol:PxEthTest
[PASS] testBurn(uint224) (runs: 101, : 93853, ~: 93874)
[PASS] testCannotBurnNoBurnerRole() (gas: 12018)
[PASS] testCannotBurnZeroAddress() (gas: 37092)
[PASS] testCannotBurnZeroAmount() (gas: 37029)
[PASS] testCannotMintNoMinterRole() (gas: 12029)
[PASS] testCannotMintZeroAddress() (gas: 36949)
[PASS] testCannotMintZeroAmount() (gas: 37084)
[PASS] testCannotOperatorApproveZeroAddress() (gas: 38918)
[PASS] testMint(uint224) (runs: 101, : 90662, ~: 90662)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 42.37ms (28.39ms CPU time)

Ran 22 tests for test/AutoPxEth.t.sol:AutoPxEthTest
[PASS] testAssetsPerShare() (gas: 30621)
[PASS] testBeforeWithdrawAssetsGreaterThanTotalStaked() (gas: 319558)
[PASS] testCannotSetPirexEthUnauthorised() (gas: 34310)
[PASS] testCannotSetPirexEthZeroAddress() (gas: 13666)
[PASS] testCannotSetPlatformFeeUnauthorised() (gas: 34254)
[PASS] testCannotSetPlatformUnauthorised() (gas: 34318)
[PASS] testCannotSetPlatformZeroAddress() (gas: 13709)
[PASS] testCannotSetSetPlatformFeeGreaterThanMaxPlatformFee() (gas: 13539)
[PASS] testCannotSetWithdrawPenaltyGreaterThanMaxWithdrawalPenalty() (gas: 13604)
[PASS] testCannotSetWithdrawalPenaltyUnauthorised() (gas: 34340)
[PASS] testDeposit(uint96) (runs: 100, : 179660, ~: 179660)
[PASS] testDepositWithPendingRewards(uint96,uint96,uint32) (runs: 100, : 249188, ~: 249188)
[PASS] testPreviewWithdraw() (gas: 140032)
[PASS] testRevertNotifyRewardAmountUnathorised() (gas: 32692)
[PASS] testRevertNotifyRewardAmountnoRewards() (gas: 46616)
[PASS] testSetPirexEth() (gas: 19840)
[PASS] testSetPlatform() (gas: 22722)
[PASS] testSetPlatformFee() (gas: 22295)
[PASS] testSetWithdrawalPenalty() (gas: 22413)
[PASS] testTransfer() (gas: 155379)
[PASS] testTransferFrom() (gas: 177117)
[PASS] testVaultHarvest(uint96,uint32) (runs: 100, : 251084, ~: 251007)
Suite result: ok. 22 passed; 0 failed; 0 skipped; finished in 200.40ms (185.51ms CPU time)

Ran 4 tests for test/PirexFees.t.sol:PirexFeesTest
[PASS] testCannotSetRecipientNotAuthorized() (gas: 11609)
[PASS] testCannotSetRecipientZeroAddress() (gas: 8473)
[PASS] testDistributeFees(uint96) (runs: 101, : 169707, ~: 169707)
[PASS] testSetRecipient() (gas: 19123)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 25.19ms (19.23ms CPU time)

Ran 29 tests for test/ValidatorQueue.t.sol:ValidatorQueueTest
[PASS] testAddInitializedValidators() (gas: 4945431)
[PASS] testCannotAddInitializedValidatorsDepositEtherUnpaused() (gas: 21490)
[PASS] testCannotAddInitializedValidatorsUnauthorized() (gas: 35868)
[PASS] testCannotAddUsedInitializedValidators() (gas: 707810)
[PASS] testCannotClearInitializedValidatorsDepositEtherUnpaused() (gas: 8488)
[PASS] testCannotClearInitializedValidatorsUnauthorized() (gas: 22927)
[PASS] testCannotGetNextValidatorQueueEmpty() (gas: 20964)
[PASS] testCannotPopInitializedValidatorOutOfBounds(uint256) (runs: 101, : 5008740, ~: 5069553)
[PASS] testCannotPopInitializedValidatorsDepositEtherUnpaused() (gas: 8493)
[PASS] testCannotPopInitializedValidatorsUnauthorized() (gas: 22994)
```

```
[PASS] testCannotPopValidatorQueueEmpty() (gas: 11501)
[PASS] testCannotRemoveInitializedMismatchValidators(uint256,bool) (runs: 101, : 5227093, ~: 5227097)
[PASS] testCannotRemoveInitializedValidatorsDepositEtherUnpaused() (gas: 9359)
[PASS] testCannotRemoveInitializedValidatorsUnauthorized() (gas: 23760)
[PASS] testCannotRemoveOrderedOutOfBounds() (gas: 10961)
[PASS] testCannotRemoveUnorderedOutOfBounds() (gas: 11102)
[PASS] testCannotSwapInitializedValidatorsDepositingEtherUnpaused() (gas: 8632)
[PASS] testCannotSwapInitializedValidatorsInvalidIndex() (gas: 19353)
[PASS] testCannotSwapInitializedValidatorsUnauthorized() (gas: 23111)
[PASS] testCannotSwapOutOfBounds() (gas: 455113)
[PASS] testCannotSwapValidatorQueueEmpty() (gas: 8600)
[PASS] testClearInitializedValidator() (gas: 5217295)
[PASS] testGetInitializedValidatorAt(uint8) (runs: 101, : 5241650, ~: 5241650)
[PASS] testGetInitializedValidatorCount() (gas: 5226350)
[PASS] testGetStakingValidatorAt(uint8) (runs: 101, : 8539785, ~: 8539785)
[PASS] testPopInitializedValidator(uint256) (runs: 101, : 5287872, ~: 5218495)
[PASS] testRemoveInitializedValidator(uint256,bool) (runs: 101, : 5141683, ~: 5164944)
[PASS] testStakingValidatorCount(uint8) (runs: 101, : 6005489, ~: 5727418)
[PASS] testSwapInitializedValidator(uint256,uint256) (runs: 100, : 5288626, ~: 5288626)

Suite result: ok. 29 passed; 0 failed; 0 skipped; finished in 33.84s (28.73s CPU time)

Ran 8 tests for test/RewardRecipientGateway.t.sol:RewardRecipientGatewayTest
[PASS] testCannotDissolveValidatorUnauthorised() (gas: 17653)
[PASS] testCannotHarvestUnauthorised() (gas: 17038)
[PASS] testCannotSlashValidatorNoEthAllowed() (gas: 22256)
[PASS] testCannotSlashValidatorUnauthorised() (gas: 18429)
[PASS] testDissolveValidator(uint96,bool) (runs: 100, : 6727924, ~: 6164673)
[PASS] testHarvest(uint96,uint96,uint96,uint8,uint96) (runs: 100, : 828622, ~: 930586)
[PASS] testReceiveEther() (gas: 18113)
[PASS] testSlashValidator(uint96,uint96,bool,bool,bool) (runs: 101, : 9294854, ~: 8742265)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 61.44s (27.54s CPU time)

Ran 9 tests for test/RewardRecipient.t.sol:RewardRecipientTest
[PASS] testCannotDissolveValidatorUnauthorised() (gas: 11604)
[PASS] testCannotHarvestUnauthorised() (gas: 11974)
[PASS] testCannotSetContractUnauthorized() (gas: 15575)
[PASS] testCannotSetContractUnrecognised() (gas: 10550)
[PASS] testCannotSetContractZeroAddress() (gas: 8979)
[PASS] testCannotSlashValidatorNoEthAllowed() (gas: 17169)
[PASS] testCannotSlashValidatorUnauthorised() (gas: 13288)
[PASS] testReceiveEther() (gas: 13217)
[PASS] testSetContract() (gas: 23737)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 61.44s (490.79µs CPU time)

Ran 101 tests for test/InstitutionalPirexEth.t.sol:InstitutionalPirexEthTest
[PASS] testBulkRedeemWithIUpxEth(uint96,uint32,bool) (runs: 100, : 7340353, ~: 6651385)
[PASS] testCannotBulkRedeemWithIUpxEthEmptyArray() (gas: 29074)
[PASS] testCannotBulkRedeemWithIUpxEthMismatchArrayLength() (gas: 29072)
[PASS] testCannotBulkRedeemWithIUpxEthNotDissolved() (gas: 35449)
[PASS] testCannotBulkRedeemWithIUpxEthPaused() (gas: 25945)
[PASS] testCannotBulkRedeemWithIUpxEthZeroAddress() (gas: 30715)
[PASS] testCannotBulkRedeemWithIUpxEthZeroAmount() (gas: 30636)
[FAIL. Reason: call did not revert as expected] testCannotDepositIncorrectValidatorParam() (gas: 1018422)
[PASS] testCannotDepositPaused() (gas: 31186)
[PASS] testCannotDepositPrivilegedDepositEtherPaused() (gas: 38247)
[PASS] testCannotDepositPrivilegedUnauthorized() (gas: 26392)
[PASS] testCannotDepositReusePubKeyForInitializedValidator() (gas: 1264362)
[PASS] testCannotDepositUnauthorised() (gas: 24347)
[PASS] testCannotDepositZeroAddress() (gas: 36330)
[PASS] testCannotDepositZeroAmount() (gas: 27786)
[PASS] testCannotDissolveValidatorInvalidAmount() (gas: 263481)
[PASS] testCannotDissolveValidatorNotWithdrawable() (gas: 265920)
[PASS] testCannotDissolveValidatorUnathorized() (gas: 225455)
[PASS] testCannotEmergencyWithdrawDepositEtherNotPaused() (gas: 16781)
[PASS] testCannotEmergencyWithdrawInvalidToken() (gas: 41961)
[PASS] testCannotEmergencyWithdrawNotPaused() (gas: 32888)
[PASS] testCannotEmergencyWithdrawUnauthorised() (gas: 22223)
```

```
[PASS] testCannotEmergencyWithdrawZeroAddress() (gas: 41903)
[PASS] testCannotEmergencyWithdrawZeroAmount() (gas: 41947)
[PASS] testCannotHarvestUnauthorized() (gas: 13526)
[PASS] testCannotInitiateRedemptionNotEnoughValidators() (gas: 9929323)
[PASS] testCannotInititateRedemptionNoPartialInitiateRedemption() (gas: 11735066)
[PASS] testCannotInititateRedemptionNoValidatorExit() (gas: 5801172)
[PASS] testCannotInititateRedemptionPaused() (gas: 24762)
[PASS] testCannotInititateRedemptionZeroAddress() (gas: 26184)
[PASS] testCannotInititateRedemptionZeroAmount() (gas: 26148)
[PASS] testCannotRedeemWithIPxEthBuffer() (gas: 489870)
[PASS] testCannotRedeemWithIPxEthPaused() (gas: 24686)
[PASS] testCannotRedeemWithIPxEthTransferFailure() (gas: 390535)
[PASS] testCannotRedeemWithIPxEthZeroAddress() (gas: 25523)
[PASS] testCannotRedeemWithIPxEthZeroAmount() (gas: 25486)
[PASS] testCannotRedeemWithIUpxEthEthTransferFailed() (gas: 5910993)
[PASS] testCannotRedeemWithIUpxEthNotDissolved() (gas: 30308)
[PASS] testCannotRedeemWithIUpxEthNotEnoughETH() (gas: 5895835)
[PASS] testCannotRedeemWithIUpxEthPaused() (gas: 24718)
[PASS] testCannotRedeemWithIUpxEthZeroAddress() (gas: 25573)
[PASS] testCannotRedeemWithIUpxEthZeroAmount() (gas: 25580)
[PASS] testCannotRemoveInitializedMismatchValidators(uint256,bool) (runs: 101, : 5348986, ~: 5348989)
[PASS] testCannotSetContractUnauthorized() (gas: 47843)
[PASS] testCannotSetContractUnrecognised() (gas: 22597)
[PASS] testCannotSetContractZeroAddress() (gas: 20854)
[PASS] testCannotSetFeeGreaterThanDenominator(uint8) (runs: 101, : 23776, ~: 23776)
[PASS] testCannotSetFeeInvalidFee(uint8) (runs: 101, : 25340, ~: 25340)
[PASS] testCannotSetFeeInvalidMaxFee(uint8) (runs: 101, : 53073, ~: 53073)
[PASS] testCannotSetFeeUnauthorized() (gas: 21611)
[PASS] testCannotSetMaxBufferPctExceedsMax() (gas: 22357)
[PASS] testCannotSetMaxBufferPctUnauthorized() (gas: 21504)
[PASS] testCannotSetMaxFeeUnauthorized(uint8) (runs: 101, : 22152, ~: 22152)
[PASS] testCannotSetMaxProcessedValidatorCountInvalidValue() (gas: 20063)
[PASS] testCannotSetMaxProcessedValidatorCountUnauthorized() (gas: 21375)
[PASS] testCannotSlashMismatchValidator(uint96,uint96,bool,bool) (runs: 101, : 9809181, ~: 9809183)
[PASS] testCannotSlashValidatorAccountNotApproved(uint96,uint96,bool) (runs: 101, : 9855368, ~: 9855370)
[PASS] testCannotSlashValidatorInvalidAmount(uint96,uint96,bool) (runs: 101, : 9816586, ~: 9816588)
[PASS] testCannotSlashValidatorNotEnoughBuffer(uint96,uint96,bool) (runs: 101, : 9858653, ~: 9858657)
[PASS] testCannotSlashValidatorStatusNotWithdrawableOrStaking(uint96,uint96,bool) (runs: 101, : 5382984, ~: 5382984)
[PASS] testCannotSlashValidatorSumMismatch(uint96,uint96,bool) (runs: 101, : 9858628, ~: 9858630)
[PASS] testCannotSlashValidatorUnathorized() (gas: 22743)
[PASS] testCannotToggleBurnerAccountsUnauthorised() (gas: 24334)
[PASS] testCannotToggleDepositEtherUnathorized() (gas: 21247)
[PASS] testCannotTogglePauseStateUnauthorized() (gas: 21294)
[PASS] testCannotTopUpStakeAccountNotApproved(uint8,uint96) (runs: 101, : 9080323, ~: 9183748)
[PASS] testCannotTopUpStakeInvalidParams() (gas: 5807911)
[PASS] testCannotTopUpStakeNoEth(uint8,uint96) (runs: 101, : 8812322, ~: 9081147)
[PASS] testCannotTopUpStakeNoEthAllowed(uint8,uint96) (runs: 101, : 9077323, ~: 9185909)
[PASS] testCannotTopUpStakeNotEnoughBuffer(uint8,uint96) (runs: 101, : 9091643, ~: 9448312)
[PASS] testCannotTopUpStakeSumMismatch(uint8,uint8,uint96) (runs: 100, : 9075544, ~: 9206257)
[PASS] testCannotTopUpStakeUnauthorized() (gas: 38470)
[PASS] testCannotTopUpStakeValidatorNotStaking() (gas: 42478)
[PASS] testClearInitializedValidator() (gas: 5341104)
[PASS] testDeposit(uint96,uint32) (runs: 100, : 445971, ~: 456111)
[PASS] testDepositPaused() (gas: 27373)
[PASS] testDepositPrivileged(uint96) (runs: 101, : 5662200, ~: 5662200)
[PASS] testDepositWithBufferNoFee(uint96,uint96) (runs: 100, : 5724262, ~: 5725077)
[PASS] testDissolveValidator(uint96) (runs: 100, : 6817879, ~: 6358302)
[PASS] testEmergencyWithdraw(uint96) (runs: 101, : 2078459, ~: 2078459)
[PASS] testHarvest(uint96,uint96,uint96,uint96) (runs: 100, : 8105044, ~: 9990855)
[PASS] testInitiateRedemptionByIPxEth(uint96,uint32,bool) (runs: 100, : 7217024, ~: 6451134)
[PASS] testInstantRedeemWithIPxEth(uint96,uint8,uint32,uint32) (runs: 100, : 641180, ~: 655424)
[PASS] testPopInitializedValidator(uint256) (runs: 101, : 5435280, ~: 5361801)
[PASS] testRedeemWithIUpxEth(uint96,uint32,bool) (runs: 100, : 7165896, ~: 6434709)
[PASS] testRemoveInitializedValidator(uint256,bool) (runs: 101, : 5262587, ~: 5230871)
```

```
[PASS] testRevertEmergencyWithdrawEthTransferFailure() (gas: 72883)
[PASS] testSetContract() (gas: 184251)
[PASS] testSetFee(uint8,uint32) (runs: 100, : 46840, ~: 51019)
[PASS] testSetMaxBufferSizePct(uint256) (runs: 101, : 91800, ~: 93574)
[PASS] testSetMaxFee(uint8,uint32) (runs: 100, : 33481, ~: 34441)
[PASS] testSetMaxProcessedValidatorCount(uint8) (runs: 101, : 58768, ~: 58768)
[PASS] testSlashValidator(uint96,uint96,bool,bool) (runs: 101, : 10066619, ~: 10138476)
[PASS] testSlashValidatorWhenExiting(uint96,bool,bool) (runs: 101, : 10278842, ~: 10371128)
[PASS] testSwapInitializedValidator(uint256,uint256) (runs: 100, : 5413728, ~: 5413728)
[PASS] testToggleBurnerAccounts() (gas: 42053)
[PASS] testTogglePauseDepositEther() (gas: 64053)
[PASS] testTogglePauseState() (gas: 27353)
[FAIL. Reason: EvmError: Revert; counterexample: calldata=0xb97902a8... args=[1, 50000 [5e4], false]]
↪  testTopUpStake(uint8,uint96,bool) (runs: 0, : 0, ~: 0)
[FAIL. Reason: EvmError: Revert] testValidatorSpinup() (gas: 418173)
[FAIL. Reason: EvmError: Revert; counterexample: calldata=0xa38860e1... args=[1]] testValidatorSpinupWithLimit(uint8)
↪  (runs: 0, : 0, ~: 0)
Suite result: FAILED. 97 passed; 4 failed; 0 skipped; finished in 61.44s (130.42s CPU time)

Ran 95 tests for test/PirexEth.t.sol:PirexEthTest
[PASS] testBulkRedeemWithUpxEth(uint96,uint32,bool) (runs: 100, : 6360936, ~: 5679232)
[PASS] testCannotBulkRedeemWithUpxEthNotDissolved() (gas: 20079)
[PASS] testCannotBulkRedeemWithUpxEthPaused() (gas: 17411)
[PASS] testCannotBulkRedeemWithUpxEthZeroAddress() (gas: 15397)
[PASS] testCannotBulkRedeemWithUpxEthZeroAmount() (gas: 15296)
[FAIL. Reason: call did not revert as expected] testCannotDepositIncorrectValidatorParam() (gas: 710579)
[PASS] testCannotDepositPaused() (gas: 22952)
[PASS] testCannotDepositPrivilegedDepositEtherPaused() (gas: 22994)
[PASS] testCannotDepositPrivilegedUnauthorized() (gas: 20736)
[PASS] testCannotDepositReusePubKeyForInitializedValidator() (gas: 759265)
[PASS] testCannotDepositZeroAddress() (gas: 22515)
[PASS] testCannotDepositZeroAmount() (gas: 13950)
[PASS] testCannotDissolveValidatorInvalidAmount() (gas: 241628)
[PASS] testCannotDissolveValidatorNotWithdrawable() (gas: 243860)
[PASS] testCannotDissolveValidatorUnathorized() (gas: 220870)
[PASS] testCannotEmergencyWithdrawDepositEtherNotPaused() (gas: 11351)
[PASS] testCannotEmergencyWithdrawInvalidToken() (gas: 24717)
[PASS] testCannotEmergencyWithdrawNotPaused() (gas: 17152)
[PASS] testCannotEmergencyWithdrawUnauthorised() (gas: 16226)
[PASS] testCannotEmergencyWithdrawZeroAddress() (gas: 22544)
[PASS] testCannotEmergencyWithdrawZeroAmount() (gas: 22543)
[PASS] testCannotHarvestUnauthorized() (gas: 8583)
[PASS] testCannotInitiateRedemptionNotEnoughValidators() (gas: 8446651)
[PASS] testCannotInitiateRedemptionNoPartialInitiateRedemption() (gas: 8673441)
[PASS] testCannotInitiateRedemptionNoValidatorExit() (gas: 8445267)
[PASS] testCannotInitiateRedemptionPaused() (gas: 16269)
[PASS] testCannotInitiateRedemptionZeroAddress() (gas: 14051)
[PASS] testCannotInitiateRedemptionZeroAmount() (gas: 13993)
[PASS] testCannotRedeemWithPxEthBuffer() (gas: 132335)
[PASS] testCannotRedeemWithPxEthPaused() (gas: 16086)
[PASS] testCannotRedeemWithPxEthTransferFailure() (gas: 188266)
[PASS] testCannotRedeemWithPxEthZeroAddress() (gas: 13825)
[PASS] testCannotRedeemWithPxEthZeroAmount() (gas: 13832)
[PASS] testCannotRedeemWithUpxEthEthTransferFailed() (gas: 5466458)
[PASS] testCannotRedeemWithUpxEthNotDissolved() (gas: 18621)
[PASS] testCannotRedeemWithUpxEthNotEnoughETH() (gas: 5492771)
[PASS] testCannotRedeemWithUpxEthPaused() (gas: 16165)
[PASS] testCannotRedeemWithUpxEthZeroAddress() (gas: 13919)
[PASS] testCannotRedeemWithUpxEthZeroAmount() (gas: 13904)
[PASS] testCannotSetContractUnauthorized() (gas: 41730)
[PASS] testCannotSetContractUnrecognised() (gas: 10865)
[PASS] testCannotSetContractZeroAddress() (gas: 9167)
[PASS] testCannotSetFeeGreaterThanDenominator(uint8) (runs: 101, : 12088, ~: 12088)
[PASS] testCannotSetFeeInvalidFee(uint8) (runs: 101, : 13212, ~: 13212)
[PASS] testCannotSetFeeInvalidMaxFee(uint8) (runs: 101, : 38675, ~: 38675)
[PASS] testCannotSetFeeUnauthorized() (gas: 16035)
[PASS] testCannotSetMaxBufferPctExceedsMax() (gas: 11026)
[PASS] testCannotSetMaxBufferPctUnauthorized() (gas: 15842)
[PASS] testCannotSetMaxFeeUnauthorized(uint8) (runs: 101, : 16708, ~: 16708)
[PASS] testCannotSetMaxProcessedValidatorCountInvalidValue() (gas: 8860)
[PASS] testCannotSetMaxProcessedValidatorCountUnauthorized() (gas: 15823)
[PASS] testCannotSlashMismatchValidator(uint96,uint96,bool,bool) (runs: 101, : 8549278, ~: 8549281)
[PASS] testCannotSlashValidatorAccountNotApproved(uint96,uint96,bool) (runs: 101, : 8592879, ~: 8592881)
```

```
[PASS] testCannotSlashValidatorInvalidAmount(uint96,uint96,bool) (runs: 101, : 8556310, ~: 8556312)
[PASS] testCannotSlashValidatorNotEnoughBuffer(uint96,uint96,bool) (runs: 101, : 8594844, ~: 8594848)
[PASS] testCannotSlashValidatorStatusNotWithdrawableOrStaking(uint96,uint96,bool) (runs: 101, : 5264965, ~: 5264965)
[PASS] testCannotSlashValidatorSumMismatch(uint96,uint96,bool) (runs: 101, : 8594872, ~: 8594876)
[PASS] testCannotSlashValidatorUnathorized() (gas: 17103)
[PASS] testCannotToggleBurnerAccountsUnauthorised() (gas: 18747)
[PASS] testCannotToggleDepositEtherUnathorized() (gas: 15742)
[PASS] testCannotTogglePauseStateUnauthorized() (gas: 15745)
[PASS] testCannotTopUpStakeAccountNotApproved(uint8,uint96) (runs: 101, : 7854963, ~: 7912146)
[PASS] testCannotTopUpStakeInvalidParams() (gas: 5600855)
[PASS] testCannotTopUpStakeNoEth(uint8,uint96) (runs: 101, : 7701501, ~: 7885219)
[PASS] testCannotTopUpStakeNoEthAllowed(uint8,uint96) (runs: 101, : 7860656, ~: 8121926)
[PASS] testCannotTopUpStakeNotEnoughBuffer(uint8,uint96) (runs: 101, : 7853897, ~: 7911073)
[PASS] testCannotTopUpStakeSumMismatch(uint8,uint8,uint96) (runs: 100, : 7886044, ~: 8040746)
[PASS] testCannotTopUpStakeUnauthorized() (gas: 32802)
[PASS] testCannotTopUpStakeValidatorNotStaking() (gas: 28186)
[PASS] testDeposit(uint96,uint32,bool,uint32) (runs: 100, : 319924, ~: 281438)
[PASS] testDepositPaused() (gas: 18237)
[PASS] testDepositPrivileged(uint96) (runs: 101, : 5364878, ~: 5362909)
[PASS] testDepositWithBufferNoFeeNoCompound(uint96,uint96) (runs: 100, : 5426095, ~: 5426514)
[PASS] testDissolveValidator(uint96) (runs: 100, : 6679227, ~: 5865730)
[PASS] testEmergencyWithdraw() (gas: 1550046)
[PASS] testHarvest(uint96,uint96,uint96) (runs: 100, : 7479172, ~: 8701025)
[PASS] testInitiateRedemptionByApxEthTransfer(uint96,uint32) (runs: 100, : 6948796, ~: 5837193)
[PASS] testInitiateRedemptionByApxEthTransferFrom(uint96,uint32) (runs: 100, : 6466423, ~: 5735391)
[PASS] testInitiateRedemptionByPxEth(uint96,uint32,bool) (runs: 100, : 6393462, ~: 5675103)
[PASS] testInstantRedeemWithPxEth(uint96,uint8,uint32,uint32) (runs: 100, : 304639, ~: 312829)
[PASS] testRedeemWithUpxEth(uint96,uint32,bool) (runs: 100, : 6344730, ~: 5677758)
[PASS] testRevertEmergencyWithdrawEthTransferFailure() (gas: 42779)
[PASS] testSetContract() (gas: 79067)
[PASS] testSetFee(uint8,uint32) (runs: 100, : 35757, ~: 38941)
[PASS] testSetMaxBufferSizePct(uint256) (runs: 101, : 34975, ~: 36946)
[PASS] testSetMaxFee(uint8,uint32) (runs: 100, : 20025, ~: 20697)
[PASS] testSetMaxProcessedValidatorCount(uint8) (runs: 101, : 17693, ~: 17693)
[PASS] testSlashValidator(uint96,uint96,bool,bool) (runs: 101, : 8666161, ~: 8672632)
[PASS] testSlashValidatorWhenExiting(uint96,bool,bool) (runs: 101, : 8676103, ~: 8687688)
[PASS] testToggleBurnerAccounts() (gas: 32452)
[PASS] testTogglePauseDepositEther() (gas: 19437)
[PASS] testTogglePauseState() (gas: 18195)
[FAIL. Reason: EvmError: Revert; counterexample: calldata=0xb97902a800... args=[1, 50000 [5e4], false]]
↪  testTopUpStake(uint8,uint96,bool) (runs: 0, : 0, ~: 0)
[FAIL. Reason: EvmError: Revert] testValidatorSpinup() (gas: 340428)
[FAIL. Reason: EvmError: Revert; counterexample: calldata=0xa38860e1... args=[1]] testValidatorSpinupWithLimit(uint8)
↪  (runs: 0, : 0, ~: 0)
Suite result: FAILED. 91 passed; 4 failed; 0 skipped; finished in 61.41s (163.01s CPU time)

Ran 10 test suites in 61.45s (279.87s CPU time): 279 tests passed, 8 failed, 0 skipped (287 total tests)

Failing tests:
Encountered 4 failing tests in test/InstitutionalPirexEth.t.sol:InstitutionalPirexEthTest
[FAIL. Reason: call did not revert as expected] testCannotDepositIncorrectValidatorParam() (gas: 1018422)
[FAIL. Reason: EvmError: Revert; counterexample: calldata=0xb97902a80... args=[1, 50000 [5e4], false]]
↪  testTopUpStake(uint8,uint96,bool) (runs: 0, : 0, ~: 0)
[FAIL. Reason: EvmError: Revert] testValidatorSpinup() (gas: 418173)
[FAIL. Reason: EvmError: Revert; counterexample: calldata=0xa38860e100... args=[1]] testValidatorSpinupWithLimit(uint8)
↪  (runs: 0, : 0, ~: 0)

Encountered 4 failing tests in test/PirexEth.t.sol:PirexEthTest
[FAIL. Reason: call did not revert as expected] testCannotDepositIncorrectValidatorParam() (gas: 710579)
[FAIL. Reason: EvmError: Revert; counterexample: calldata=0xb97902a800000... args=[1, 50000 [5e4], false]]
↪  testTopUpStake(uint8,uint96,bool) (runs: 0, : 0, ~: 0)
[FAIL. Reason: EvmError: Revert] testValidatorSpinup() (gas: 340428)
[FAIL. Reason: EvmError: Revert; counterexample:
↪  calldata=0xa38860e1000000000000000000000000000000000000000000000000000000000000000001 args=[1]]
↪  testValidatorSpinupWithLimit(uint8) (runs: 0, : 0, ~: 0)

Encountered a total of 8 failing tests, 279 tests succeeded
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.