BLOCKSEC

Security Audit

Report for puffer-institutional

Date: August 6, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Puffer
Target	puffer-institutional

Version History

Version	Date	Description
1.0	August 6, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-08-06 13:00:29+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of puffer-institutional of Puffer.

The Puffer Institutional Vault is a permissioned, upgradeable ETH staking vault that uses ERC-4626-style share accounting and supports both restaked and non-restaked Beacon Chain validators. Oracle-reported validator balances are combined with the vault's liquid ETH and WETH holdings to determine the share exchange rate. This upgrade adds performance fee settlement to the Oracle reporting flow. Rewards are derived from changes in total assets after excluding net LP flows and are checked against an APR circuit breaker before fee shares are minted. It also introduces OracleReportSanityChecker to bound validator balance updates and report frequency, and adds native ETH withdrawal and redemption support.

Note this audit only focuses on the smart contracts in the following directories/files. Code prior to and including the baseline version 0, where applicable, is outside the scope of this audit and assumes to be reliable and secure.

- src/InstitutionalVault.sol
- src/InstitutionalVaultStorage.sol
- src/NonRestakingWithdrawalCredentials.sol
- src/OracleReportSanityChecker.sol
- src/interface/IInstitutionalVault.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
puffer-institutional	`Version 0`	`2cc12d96bcbf9b207f41cfbdeb0b9c64312ee628`
	`Version 1`	`23e28359b763388ec988122b640e81d05e754429`
	`Version 2`	`42246ecda45664d3c57475641468e431cc35f806`

[1] https://github.com/PufferFinance/puffer-institutional

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation

- Error-prone randomness
- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **five** recommendations and **nine** notes.

- Low Risk: 3
- Recommendation: 5
- Note: 9

ID	Severity	Description	Category	Status
1	Low	Incorrect reward accounting	Security Issue	Confirmed
2	Low	Potential front-running attack	Security Issue	Confirmed
3	Low	Incorrect accounting of `nonRestakedValidatorsETH`	Security Issue	Confirmed
4	-	Align documentation with implementation	Recommendation	Fixed
5	-	Add non-zero checks on deposits and withdrawals	Recommendation	Fixed
6	-	Emit native exit events	Recommendation	Fixed
7	-	Add a non-zero check on the `receiver`	Recommendation	Fixed
8	-	Validate the deposit amount and public key length	Recommendation	Confirmed
9	-	Potential centralization risks	Note	-
10	-	Share price dependence on reported validator balances	Note	-
11	-	Atomic upgrade and initialization	Note	-
12	-	Withdrawal error selector compatibility	Note	-
13	-	Authority migration for the withdrawal credentials contract	Note	-
14	-	Trust assumptions of Oracle roles	Note	-
15	-	Ensure pending rewards are reported	Note	-
16	-	Fee mechanisms are controlled by institutions	Note	-
17	-	Ensure correct handling of validator consolidations	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Incorrect reward accounting

Severity Low

Status Confirmed

Introduced by Version 1

Description In the contract `InstitutionalVault`, functions `_withdraw()` and `_exitNativeETH()` decrease `netLPFlowSinceLastReport` by the nominal withdrawal amount. When the vault is the receiver of a WETH withdrawal, the self-transfer does not change its WETH balance. For ETH withdrawals, the receiver can return ETH to the vault in its callback. In both cases, shares are burned, and `netLPFlowSinceLastReport` is reduced even though the withdrawn assets remain in the vault. Consequently, the next `_accrueFees()` execution treats these retained assets as rewards.

An attacker can temporarily deposit assets to obtain most of the vault's shares, create an artificial reward through either withdrawal function, and then recover most of the temporary liquidity through a normal withdrawal. The attacker bears only the portion of the artificial reward diluted by prior depositors, while the manipulated `netLPFlowSinceLastReport` persists. Moreover, function `rescueAnything()` does not correct the manipulated `netLPFlowSinceLastReport`.

```
893         address caller = _msgSender();
894
895         if (caller != owner) {
896             _spendAllowance(owner, caller, shares);
897         }
898
899         _burn(owner, shares);
900
901         _decreaseNetLPFlow(assets);
902
903         _unwrapWETH(assets);
904
905         (bool success,) = receiver.call{ value: assets }("");
906         require(success, EthTransferFailed());
907
908         emit Withdraw(caller, receiver, owner, assets, shares);
909     }
```

Listing 2.1: src/InstitutionalVault.sol

```
1002        internal
1003        virtual
1004        override
1005    {
1006        super._withdraw(caller, receiver, owner, assets, shares);
1007        _decreaseNetLPFlow(assets);
1008    }
```

Listing 2.2: src/InstitutionalVault.sol

```
808         uint256 newTA = totalAssets();
809
810         int256 rawDelta = int256(newTA) - int256(uint256($.lastTotalAssets));
811         int256 rewards = rawDelta - int256($.netLPFlowSinceLastReport);
812
813         if (rewards > 0) {
814             uint256 r = uint256(rewards);
```

```
815         uint256 elapsed = block.timestamp - uint256($.lastReportTimestamp);
816         uint256 maxRewards =
817             (uint256($.maxAprBps) * uint256($.lastTotalAssets) * elapsed) / SECONDS_PER_YEAR /
                BPS_DENOMINATOR;
818         // Circuit breaker: rewards above the APR cap revert; the admin raises maxAprBps to
                clear a genuine spike.
819         // Skipped below MIN_APR_BREAKER_BASELINE, where the percentage bound can be smaller
                than a real reward.
820         if ($.lastTotalAssets > MIN_APR_BREAKER_BASELINE) {
821             require(r <= maxRewards, RewardsExceedSanityBound());
822         }
823         _mintFeeShares($, r, newTA);
824     }
```

Listing 2.3: src/InstitutionalVault.sol

Impact Attackers can create artificial rewards and may cause oracle reports to revert when the cap is exceeded.

Suggestion Revise the code logic accordingly.

Feedback from the project The project confirms this as an accounting inaccuracy and will not apply code changes. Nobody outside the institution can call withdraw in this deployment, and even if the function were opened to everyone, the caller could not profit from it.

2.1.2 Potential front-running attack

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `InstitutionalVault`, the function `setValidatorsETH()` updates the validator ETH. When it reports an increase in validator balances, `totalAssets()` and the share price increase. According to the audit documentation, an institution can open the deposit and withdrawal functions to everyone. When these functions are public, anyone can observe a pending `setValidatorsETH()` transaction and profit through a front-running attack. Specifically, an attacker deposits right before the report to acquire shares at the pre-report exchange rate, then redeems after the report to capture a portion of the rewards.

```
537   function setValidatorsETH(uint128 restakedValidatorsETH, uint128 nonRestakedValidatorsETH)
538       external
539       virtual
540       restricted
541   {
542       Storage storage $ = _getVaultStorage();
543       $.restakedValidatorsETH = restakedValidatorsETH;
544       $.nonRestakedValidatorsETH = nonRestakedValidatorsETH;
545       emit RestakedValidatorsETHUpdated(restakedValidatorsETH);
546       emit NonRestakedValidatorsETHUpdated(nonRestakedValidatorsETH);
547       _accrueFees($);
548   }
```

Listing 2.4: src/InstitutionalVault.sol

Impact Depositors who front-run oracle reports can capture part of the pending validator rewards, diluting the returns of prior depositors.

Suggestion Require a minimum holding period before shares are redeemable, and submit the report through a private transaction.

Feedback from the project The project confirms this issue and will not apply code changes. In this deployment, nobody outside the institution can deposit or withdraw. If those functions are ever opened to the public, the project will mitigate the risk by submitting oracle reports through a private RPC at short intervals.

2.1.3 Incorrect accounting of `nonRestakedValidatorsETH`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `InstitutionalVault`, function `withdrawNonRestakedETH()` is expected to sweep ETH into the vault and reduce `nonRestakedValidatorsETH` only by the amount attributed to validator withdrawals. However, the function computes the reduction as the full ETH balance delta after sweeping the contract `NonRestakingWithdrawalCredentials`, including ETH unrelated to validator withdrawals. Consequently, direct donations and unused `msg.value` retained by functions `requestWithdrawal()` and `requestConsolidation()` are incorrectly treated as validator withdrawal proceeds.

```
494   function withdrawNonRestakedETH() external virtual restricted {
495       uint256 balanceBefore = _ethPlusWeth();
496       Storage storage $ = _getVaultStorage();
497       NonRestakingWithdrawalCredentials(payable($.noRestakingWithdrawalCredentials)).withdrawETH
              ();
498       uint256 balanceAfter = _ethPlusWeth();
499       uint256 difference = balanceAfter - balanceBefore;
500       // Saturating subtraction via Math.trySub to avoid an underflow panic
501       (, uint256 result) = Math.trySub($.nonRestakedValidatorsETH, SafeCast.toUint128(difference)
              );
502       uint128 newNonRestakedValidatorsETH = SafeCast.toUint128(result);
503       $.nonRestakedValidatorsETH = newNonRestakedValidatorsETH;
504       emit NonRestakedValidatorsETHUpdated(newNonRestakedValidatorsETH);
505   }
```

Listing 2.5: src/InstitutionalVault.sol

```
55   // Allow donations to the contract
56   receive() external payable { }
```

Listing 2.6: src/NonRestakingWithdrawalCredentials.sol

```
68  function requestWithdrawal(IEigenPodTypes.WithdrawalRequest[] calldata requests) external
        payable restricted {
69      uint256 fee = getWithdrawalRequestFee();
70      // The remainder is donated and not refunded to the caller
71      require(msg.value >= fee * requests.length, NotEnoughETH());
72
73      for (uint256 i = 0; i < requests.length; ++i) {
74          // We don't need to validate the length of the pubkeys as the precompile will revert if
                the pubkeys are of invalid length
75          bytes memory callData = abi.encodePacked(requests[i].pubkey, requests[i].amountGwei);
76          (bool ok,) = WITHDRAWAL_REQUEST_ADDRESS.call{ value: fee }(callData);
77          require(ok, WithdrawalRequestFailed());
78          emit WithdrawalRequested(requests[i].pubkey, requests[i].amountGwei);
79      }
80  }
```

Listing 2.7: src/NonRestakingWithdrawalCredentials.sol

Impact The tracked `nonRestakedValidatorsETH` may understate the vault's actual non-restaked validator balance.

Suggestion Decrease `nonRestakedValidatorsETH` only by explicitly tracked validator withdrawal amounts.

Feedback from the project The project confirms this issue and will not apply code changes. The oracle reports this counter as the beacon chain balance plus the credentials contract's balance, so the sweep subtracts exactly what the counter already held, resulting in no discrepancy.

2.2 Recommendation

2.2.1 Align documentation with implementation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Several descriptions in the interface `IInstitutionalVault` do not match the implementation.

1. The documentation for `maxTotalFeeBps_` (line 250) permits up to 10,000 basis points, while `HARD_FEE_BPS_CAP` limits the value to 9,900 basis points.

2. The documentation for function `completeQueuedWithdrawals()` (lines 178-179) states that, when `receiveAsTokens[i]` is enabled, `restakedValidatorsETH` is reduced by the corresponding withdrawal shares. However, the function reduces it by the balance increase of the ETH and WETH.

3. The documentation for `FeesAccrued.rewards` (line 44) describes consensus rewards or slashing. The emitted value is the change in `totalAssets()` minus `netLPFlowSinceLastReport`.

```
250     * @param maxTotalFeeBps_ The Σ-recipient-bps fee cap in basis points, in (0, 10_000].
251     */
```

```
252   function initializerV3(FeeRecipient[] calldata initialRecipients, uint16 maxAprBps_, uint16
          maxTotalFeeBps_)
253       external;
```

Listing 2.8: src/interface/IInstitutionalVault.sol

```
61    /// @notice Hard ceiling on the Σ-recipient fee, kept below 100% so the mint denominator can
          never reach 0.
62    uint16 internal constant HARD_FEE_BPS_CAP = 9_900; //99%
```

Listing 2.9: src/InstitutionalVault.sol

```
176    * @notice Completes the queued withdrawals on the EigenLayer
177    * @param withdrawals The withdrawals to complete
178    * @param receiveAsTokens Per withdrawal: true to receive the underlying ETH (and reduce
179    *       restakedValidatorsETH by its shares), false to receive redelegatable shares.
180    */
181   function completeQueuedWithdrawals(
182       IDelegationManagerTypes.Withdrawal[] calldata withdrawals,
183       bool[] calldata receiveAsTokens
184   ) external;
```

Listing 2.10: src/interface/IInstitutionalVault.sol

```
43    * @notice Event emitted at the end of each oracle report after fee accrual.
44    * @param rewards The signed rewards delta (positive = consensus rewards, negative = slashing).
45    * @param newTotalAssets The vault's totalAssets() after the report.
46    */
47   event FeesAccrued(int256 rewards, uint256 newTotalAssets);
```

Listing 2.11: src/interface/IInstitutionalVault.sol

Suggestion Align the interface comments with the implemented fee cap, withdrawal reconciliation logic, and reward accounting.

2.2.2 Add non-zero checks on deposits and withdrawals

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `InstitutionalVault`, functions `depositETH()` and `deposit()` compute the minted shares using `previewDeposit()`, which rounds down. However, neither function verifies that the resulting share amount is non-zero. Consequently, a user may deposit assets without receiving any shares.

Similarly, functions `redeem()` and `redeemETH()` compute returned assets using `previewRedeem()`, which may round down to zero. Consequently, an owner may burn shares without receiving any assets.

```
165   function depositETH(address receiver) public payable virtual restricted returns (uint256) {
166       uint256 maxAssets = maxDeposit(receiver);
167       if (msg.value > maxAssets) {
168           revert ERC4626ExceededMaxDeposit(receiver, msg.value, maxAssets);
```

```
169         }
170
171         uint256 shares = previewDeposit(msg.value);
172
173         _mint(receiver, shares);
174
175         // depositETH bypasses OZ ERC4626 _deposit; track LP flow directly.
176         _increaseNetLPFlow(msg.value);
177
178         emit Deposit(_msgSender(), receiver, msg.value, shares);
179
180         return shares;
181     }
```

Listing 2.12: src/InstitutionalVault.sol

```
238         uint256 maxShares = maxRedeem(owner);
239         if (shares > maxShares) {
240             revert ERC4626ExceededMaxRedeem(owner, shares, maxShares);
241         }
242
243         uint256 assets = previewRedeem(shares);
244
245         _exitNativeETH(assets, shares, receiver, owner);
246
247         return assets;
248     }
```

Listing 2.13: src/InstitutionalVault.sol

Suggestion Require non-zero shares and assets in the deposit and withdrawal functions, respectively.

2.2.3 Emit native exit events

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The functions `redeemETH()` and `withdrawETH()` pay native ETH but emit only the ERC-4626 `Withdraw` event. Since the vault's underlying asset is WETH, off-chain indexers may interpret this event as a WETH payout rather than a native ETH exit.

```
238         uint256 maxShares = maxRedeem(owner);
239         if (shares > maxShares) {
240             revert ERC4626ExceededMaxRedeem(owner, shares, maxShares);
241         }
242
243         uint256 assets = previewRedeem(shares);
244
245         _exitNativeETH(assets, shares, receiver, owner);
246
247         return assets;
248     }
```

Listing 2.14: src/InstitutionalVault.sol

```
287         uint256 maxAssets = maxWithdraw(owner);
288         if (assets > maxAssets) {
289             revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
290         }
291
292         uint256 shares = previewWithdraw(assets);
293
294         _exitNativeETH(assets, shares, receiver, owner);
295
296         return shares;
297     }
```

Listing 2.15: src/InstitutionalVault.sol

```
893         address caller = _msgSender();
894
895         if (caller != owner) {
896             _spendAllowance(owner, caller, shares);
897         }
898
899         _burn(owner, shares);
900
901         _decreaseNetLPFlow(assets);
902
903         _unwrapWETH(assets);
904
905         (bool success,) = receiver.call{ value: assets }("");
906         require(success, EthTransferFailed());
907
908         emit Withdraw(caller, receiver, owner, assets, shares);
909     }
```

Listing 2.16: src/InstitutionalVault.sol

Suggestion Emit a dedicated native exit event so off-chain consumers can identify the actual payout asset.

2.2.4 Add a non-zero check on the `receiver`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_exitNativeETH()` does not verify that `receiver` is a non-zero address. An ETH transfer to the zero address succeeds after the owner's shares are burned, resulting in the permanent loss of the withdrawn ETH.

```
893         address caller = _msgSender();
894
895         if (caller != owner) {
```

```
896             _spendAllowance(owner, caller, shares);
897         }
898
899         _burn(owner, shares);
900
901         _decreaseNetLPFlow(assets);
902
903         _unwrapWETH(assets);
904
905         (bool success,) = receiver.call{ value: assets }("");
906         require(success, EthTransferFailed());
907
908         emit Withdraw(caller, receiver, owner, assets, shares);
909     }
```

Listing 2.17: src/InstitutionalVault.sol

Suggestion Reject the zero address before burning shares.

2.2.5 Validate the deposit amount and public key length

Status Confirmed

Introduced by `Version 1`

Description In the contract `InstitutionalVault`, the function `_startValidators()` passes each `amountsInGwei[i]` and `pubKeys[i]` directly to `BEACON_DEPOSIT_CONTRACT.deposit()` without validating them beforehand. The beacon deposit contract requires a minimum deposit of 1 ETH and a public key of 48 bytes, so malformed inputs only revert inside the deposit contract, which wastes gas and makes the failure harder to diagnose. It is recommended to validate these inputs before calling the deposit contract.

Suggestion Add a check that each `_convertGweiToWei(amountsInGwei[i])` is at least 1 ETH and each `pubKeys[i]` is 48 bytes long.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description In the contract `InstitutionalVault`, several privileged roles can conduct sensitive operations through the `restricted` functions gated by contract `AccessManaged`, which introduces potential centralization risks. For example, the oracle reports the validator balances through `setValidatorsETH()`, and the admin configures the fee mechanism and can make arbitrary external calls with the vault funds through `customExternalCall()`. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

The authority in contract `AccessManaged` should be configured correctly, with roles and admins properly assigned and no execution delay on emergency operations such as pausing.

Meanwhile, the privileged accounts should be managed by a multi-signature wallet or a time-lock.

2.3.2 Share price dependence on reported validator balances

Introduced by Version 1

Description The two variables `_getVaultStorage().restakedValidatorsETH` and `_getVaultStorage().nonRestakedValidatorsETH` are used in the calculation of share price. These two variables are updated within the function `setValidatorsETH()` based on reported validator balances. If the reported values are inaccurate or stale, the vault's share price may also be inaccurate. The project has stated that it will address exchange rate accuracy, including the precision of share values, when the logic and design of the protocol are updated.

2.3.3 Atomic upgrade and initialization

Introduced by Version 1

Description In the contract `InstitutionalVault`, the function `initializerV3()` deploys the non-restaking withdrawal credentials contract and bootstraps the v3 fee state. If the function `setValidatorsETH()` is invoked before `initializerV3()` completes, `_accrueFees()` may execute against uninitialized fee state. For an already-deployed vault, the upgrade and `initializerV3()` should be executed in the same transaction. For a new vault, the deployment and `initializerV3()` should be executed in the same transaction.

```
141
142         // Deploy WC for fresh vaults; idempotent skip for upgraded ones.
143         if ($.noRestakingWithdrawalCredentials == address(0)) {
144             (bool success,) = address(NON_RESTAKING_WITHDRAWAL_CREDENTIALS_FACTORY).delegatecall(
145                 abi.encodeCall(NonRestakingWithdrawalCredentialsFactory.
                        deployNewStakingWithdrawalCredentials, ())
146             );
147             require(success, DelegateCallFailed());
148         }
149
150         // Bootstrap fee state. The lastTotalAssets snapshot prevents the first oracle report from
151         // minting fees on the entire pre-existing TVL.
152         $.lastTotalAssets = SafeCast.toUint128(totalAssets());
153         $.lastReportTimestamp = uint64(block.timestamp);
154         $.netLPFlowSinceLastReport = 0;
155         $.maxAprBps = maxAprBps_;
156         $.maxTotalFeeBps = maxTotalFeeBps_;
157         _setFeeRecipientsInternal($, initialRecipients);
158     }
```

Listing 2.18: src/InstitutionalVault.sol

```
537     function setValidatorsETH(uint128 restakedValidatorsETH, uint128 nonRestakedValidatorsETH)
538         external
539         virtual
540         restricted
```

```
541   {
542       Storage storage $ = _getVaultStorage();
543       $.restakedValidatorsETH = restakedValidatorsETH;
544       $.nonRestakedValidatorsETH = nonRestakedValidatorsETH;
545       emit RestakedValidatorsETHUpdated(restakedValidatorsETH);
546       emit NonRestakedValidatorsETHUpdated(nonRestakedValidatorsETH);
547       _accrueFees($);
548   }
```

Listing 2.19: src/InstitutionalVault.sol

2.3.4 Withdrawal error selector compatibility

Introduced by Version 1

Description Contract `NonRestakingWithdrawalCredentials` may revert with different error selectors across vault deployments. In baseline version 0, function `withdrawETH()` uses `Address.sendValue()` and may revert with `FailedCall()` (`0xd6bda275`). Target version 1 performs a low-level call directly and reverts with `EthTransferFailed()` (`0x6d963f88`). The function `initializerV3()` preserves existing withdrawal credentials for upgraded vaults but deploys a new version 1 instance when no instance exists. Monitoring systems and error decoders should therefore recognize both selectors across vault deployments.

```
59        require(msg.sender == vault, Unauthorized());
60        Address.sendValue(payable(vault), address(this).balance);
61    }
```

Listing 2.20: src/NonRestakingWithdrawalCredentials.sol

```
59        require(msg.sender == vault, Unauthorized());
60        (bool success,) = payable(vault).call{ value: address(this).balance }("");
61        require(success, EthTransferFailed());
62    }
```

Listing 2.21: src/NonRestakingWithdrawalCredentials.sol

```
141
142       // Deploy WC for fresh vaults; idempotent skip for upgraded ones.
143       if ($.noRestakingWithdrawalCredentials == address(0)) {
144           (bool success,) = address(NON_RESTAKING_WITHDRAWAL_CREDENTIALS_FACTORY).delegatecall(
145               abi.encodeCall(NonRestakingWithdrawalCredentialsFactory.
                      deployNewStakingWithdrawalCredentials, ())
146           );
147           require(success, DelegateCallFailed());
148       }
```

Listing 2.22: src/InstitutionalVault.sol

2.3.5 Authority migration for the withdrawal credentials contract

Introduced by Version 1

Description In the contract `NonRestakingWithdrawalCredentials`, the constructor sets the authority from the authority of the contract `InstitutionalVault` at deployment. The functions `requestWithdrawal()` and `requestConsolidation()` are guarded by this authority. When the contract `InstitutionalVault` rotates its own authority through `setAuthority()`, the credentials contract keeps its original authority, since the two authorities are set independently. The credentials contract remains operable as long as its authority is migrated together with that of the vault, so that both point to the active `AccessManager`.

2.3.6 Trust assumptions of Oracle roles

Introduced by `Version 1`

Description In the contract `InstitutionalVault`, the Oracle role reaches the vault only through the function `setValidatorsETH()`, and cannot move funds, mint shares, or change the fee and access configuration. The role is trusted for the honesty and freshness of the two reported balances. In the planned deployment, the role is held by the contract `OracleReportSanityChecker`, and reports are submitted by the account holding `SUBMITTER_ROLE`. This account should hold no other permission on the vault.

2.3.7 Ensure pending rewards are reported

Introduced by `Version 1`

Description In the contract `InstitutionalVault`, consensus layer rewards enter the vault via functions `withdrawNonRestakedETH()` and `completeQueuedWithdrawals()`. Any ETH or WETH arriving at the vault by other means are also treated as rewards. These rewards are only recognized at the next oracle report, with a portion allocated to fee recipients. If a user withdraws before that report, the user may receive an excessive share of the unreported rewards. Since the vault is controlled by the institution and not open to the public, the institution should ensure that all pending rewards are reported before submitting withdrawals.

2.3.8 Fee mechanisms are controlled by institutions

Introduced by `Version 1`

Description In the contract `InstitutionalVault`, fee recipients are configured by the admin, each assigned a fixed fee rate. Their rates are independent of contribution, performance, and whether validators are restaked or non-restaked. Additionally, the function `setFeeRecipients()` changes the fee rate without settling pending fees since the last oracle report. The project should ensure pending fees are settled before such modifications, to prevent misattribution of fees.

Feedback from the project The project stated that the fee rates are intentionally not tied to contribution. The institution that owns the vault sets the fee recipients based on its own commercial decision and may update them at any time. Because the function `setFeeRecipients()` does not settle pending fees, the project will address this operationally rather than at the contract level. They will add a required ordering, i.e., submit the oracle report before updating the recipients, into the operations guidance.

2.3.9 Ensure correct handling of validator consolidations

Introduced by Version 2

Description Version 2 introduces the function `recordValidatorPrincipal()` to handle changes in validator ETH that originate from consolidation requests. When an external validator consolidates into a vault-owned validator, the resulting increase in ETH should be accounted for as principal rather than as rewards. The corresponding shares should be minted to the contributor. Conversely, when a vault-owned validator consolidates into an external validator, the corresponding shares should be burned.

Only the admin is granted the privilege to invoke function `recordValidatorPrincipal()`, as well as `adminMint()` and `adminBurn()`, which update the shares. Because the vault provides no batching function, the admin must batch these calls. The share update and the principal record must be executed atomically in a single transaction, with the mint or burn ordered first so that the shares are priced at the exchange rate that applied before the move.

Feedback from the project The project stated that the admin of these vaults is a Safe multisig wallet, so the batched calls will be submitted as a single multisend transaction.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS