BLOCKSEC

Security Audit

Report for Puffer pufETH and PufferPool

Date: April 23, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Puffer Finance
Target	Puffer pufETH and PufferPool

Version History

Version	Date	Description
1.0	April 23, 2024	First Release

Signature

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 Target Contracts

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

The focus of this audit is on the pufETH [1] and PufferPool [2] of the Puffer Finance. The Puffer Finance is a decentralized native liquid restaking protocol built on Eigenlayer, allowing anyone to stake LST or native tokens, and run Ethereum PoS validators with enhanced rewards. It offers features like permissionless validation, native restaking, slash protection, liquid staking, restaking rewards, and more, aiming to outpace traditional liquid staking protocols while capping its growth to ensure it does not threaten Ethereum's neutrality. pufETH is a yield-bearing ERC20 token that appreciates in value as the underlying staking vault. PufferPool contains the implementation contracts of the puffer protocol.

Please note that the audit scope is limited to the following smart contracts:

- pufETH/src/PufferVaultV2.sol
- pufETH/src/PufferDepositorV2.sol
- PufferPool/src/ValidatorTicketStorage.sol
- PufferPool/src/PufferOracle.sol
- PufferPool/src/ValidatorTicket.sol
- PufferPool/src/PufferModule.sol
- PufferPool/src/GuardianModule.sol
- PufferPool/src/RestakingOperator.sol
- PufferPool/src/LibBeaconchainContract.sol
- PufferPool/src/EnclaveVerifier.sol
- PufferPool/src/PufferModuleManager.sol
- PufferPool/src/PufferProtocol.sol
- PufferPool/src/PufferOracleV2.sol
- PufferPool/src/PufferProtocolStorage.sol
- PufferPool/src/LibGuardianMessages.sol

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.

[1] https://github.com/PufferFinance/pufETH

[2] https://github.com/PufferFinance/PufferPool

Project	Version	Commit Hash
pufETH	Version 1	5db7863db529e007a43bacd88aa7809332027fae
	Version 2	0ac03edb16d61df1a61dca41722d5d1a4c634c06
Project	Version	Commit Hash
pufferpool	Version 1	d9e7948ef18f7b03c9b98999e01eeb967597879b
	Version 2	7b8b431a159fd1a847dca85c61f1eaef25315e4a

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 smart contracts, 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 smart contracts.

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 smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts 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 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling

- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 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

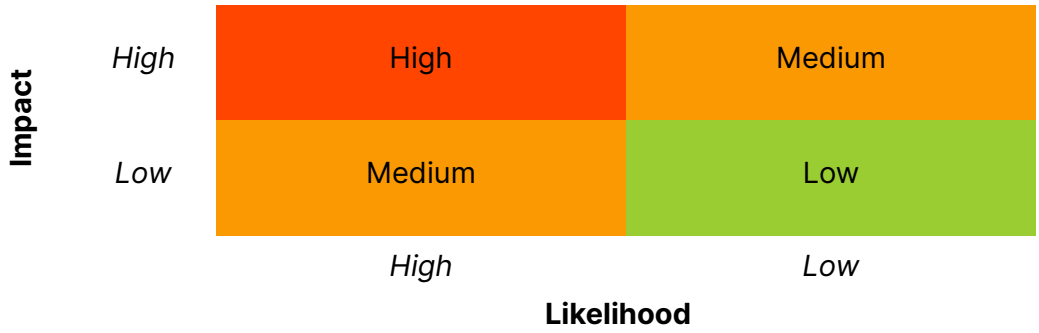
To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is 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.

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

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

Table 1.1: Vulnerability Severity Classification



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 four categories:

- **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.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **eight** potential security issues. Besides, we also have **six** recommendations and **three** notes.

- High Risk: 1
- Medium Risk: 2
- Low Risk: 5
- Recommendation: 6
- Note: 3

ID	Severity	Description	Category	Status
1	High	Lack of `VT` Penalty in Function `batchHandleWithdrawals()`	DeFi Security	Fixed
2	Medium	Signature Reuse Due to Guardian Messages Collision	DeFi Security	Confirmed
3	Low	Potential Reward Loss Due to Strict Block Number Validation	DeFi Security	Confirmed
4	Medium	Potential Inconsistency in the bondAmount Calculation	DeFi Security	Fixed
5	Low	Potential DoS attack due to the depositRootHash Verification	DeFi Security	Confirmed
6	Low	Lack of Limits Check in Function `changeMinimumVTAmount()` and `setVTPenalty()`	DeFi Security	Fixed
7	Low	Lack of Check on Node Operator Existence Description	DeFi Security	Confirmed
8	Low	Lack of Refund in the `registerValidatorKey()` Function	DeFi Security	Fixed
9	-	Typo in `docs/PufferProtocol.md`	Recommendation	Fixed
10	-	Redundant Code in the Constructor of `PufferVaultV2`	Recommendation	Confirmed
11	-	Add Range Checks in Function `initiateETHWithdrawalsFromLido()`	Recommendation	Confirmed
12	-	Remove Used `requestIds` in Function `claimWithdrawalsFromLido()`	Recommendation	Fixed
13	-	Improper Check in the Function `_setEjectionThreshold()`	Recommendation	Fixed
14	-	Improper Check in the Function `_checkValidatorRegistrationInputs()`	Recommendation	Confirmed
15	-	Validator Selection is Based on the Module instead of Register Time	Note	-
16	-	Access Control is Aligned with Function Annotations	Note	-
17	-	Potential Centralization Risks	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Lack of VT Penalty in Function batchHandleWithdrawals()

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description A malicious user can register a validator and immediately send an exit message on the beacon chain once it becomes active. Meanwhile, function batchHandleWithdrawals() only burns the validator's used VT, which is rather low, even when the validators exit before minimumVtAmount days have elapsed.

In this case, malicious users can repeatedly register and exit with multiple validators, resulting in Ether being stuck in the exit queue without earning any rewards.

```
298   function batchHandleWithdrawals(
299       StoppedValidatorInfo[] calldata validatorInfos,
300       bytes[] calldata guardianEOASignatures
301   ) external restricted {
302       GUARDIAN_MODULE.validateBatchWithdrawals(validatorInfos, guardianEOASignatures);
303
304       ProtocolStorage storage $ = _getPufferProtocolStorage();
305
306       BurnAmounts memory burnAmounts;
307       Withdrawals[] memory bondWithdrawals = new Withdrawals[](validatorInfos.length);
308
309       // We MUST NOT do the burning/oracle update/transferring ETH from the PufferModule ->
              PufferVault
310       // because it affects pufETH exchange rate
311
312       // First, we do the calculations
313       // slither-disable-start calls-loop
314       for (uint256 i = 0; i < validatorInfos.length; ++i) {
315           Validator storage validator =
316               $.validators[validatorInfos[i].moduleName][validatorInfos[i].pufferModuleIndex];
317
318           if (validator.status != Status.ACTIVE) {
319               revert InvalidValidatorState(validator.status);
320           }
321
322           // Save the Node address for the bond transfer
323           bondWithdrawals[i].node = validator.node;
324
325           // Get the burnAmount for the withdrawal at the current exchange rate
326           uint256 burnAmount =
327               _getBondBurnAmount({ validatorInfo: validatorInfos[i], validatorBondAmount:
                      validator.bond });
328           uint256 vtBurnAmount = _getVTBurnAmount(validatorInfos[i]);
329           // Update the burnAmounts
```

Listing 2.1: PufferPool/src/PufferProtocol.sol

```
798   function _getVTBurnAmount(StoppedValidatorInfo calldata validatorInfo) internal pure returns (
          uint256) {
799       uint256 validatedEpochs = validatorInfo.endEpoch - validatorInfo.startEpoch;
800       // Epoch has 32 blocks, each block is 12 seconds, we upscale to 18 decimals to get the VT
              amount and divide by 1 day
801       // The formula is validatedEpochs * 32 * 12 * 1 ether / 1 days
              (4444444444444444.44444444...) we round it up
802       return validatedEpochs * 4444444444444445;
803    }
```

Listing 2.2: PufferPool/src/PufferProtocol.sol

Impact The protocol's liquidity will be occupied, and the stakers' APR can be rather low.

Suggestion Penalize the node operators that exit before `minimumVtAmount` days have elapsed in the function `batchHandleWithdrawals()`.

2.1.2 Signature Reuse Due to Guardian Messages Collision

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description By signing different messages and posting signatures, guardians are capable of provisioning new validators, skipping malformed registrations, handling withdrawals, reporting the total number of active validators, etc.

The message digest to be signed is calculated from different types of fields. However, there is no type identifier (e.g. type hash) in these messages, which brings possible collisions for identical structures.

```
14   library LibGuardianMessages {
15       using MessageHashUtils for bytes32;
16
17       /**
18        * @notice Returns the message that the guardian's enclave needs to sign
19        * @param pufferModuleIndex is the validator index in Puffer
20        * @param signature is the BLS signature of the deposit data
21        * @param withdrawalCredentials are the withdrawal credentials for this validator
22        * @param depositDataRoot is the hash of the deposit data
23        * @return hash of the data
24        */
25       function _getBeaconDepositMessageToBeSigned(
26           uint256 pufferModuleIndex,
27           bytes memory pubKey,
28           bytes memory signature,
29           bytes memory withdrawalCredentials,
30           bytes32 depositDataRoot
31       ) internal pure returns (bytes32) {
32           return keccak256(abi.encode(pufferModuleIndex, pubKey, withdrawalCredentials, signature
                 , depositDataRoot))
33               .toEthSignedMessageHash();
```

```
34      }
35
36      /**
37       * @notice Returns the message to be signed for skip provisioning
38       * @param moduleName is the name of the module
39       * @param index is the index of the skipped validator
40       * @return the message to be signed
41       */
42      function _getSkipProvisioningMessage(bytes32 moduleName, uint256 index) internal pure
            returns (bytes32) {
43          // All guardians use the same nonce
44          return keccak256(abi.encode(moduleName, index)).toEthSignedMessageHash();
45      }
46
47      /**
48       * @notice Returns the message to be signed for handling the batch withdrawal
49       * @param validatorInfos is an array of validator information
50       * @return the message to be signed
51       */
52      function _getHandleBatchWithdrawalMessage(StoppedValidatorInfo[] memory validatorInfos)
53          internal
54          pure
55          returns (bytes32)
56      {
57          return keccak256(abi.encode(validatorInfos)).toEthSignedMessageHash();
58      }
59
60      /**
61       * @notice Returns the message to be signed updating the number of validators
62       * @param numberOfValidators is the new number of validators
63       * @param epochNumber is the epoch number
64       * @return the message to be signed
65       */
66      function _getSetNumberOfValidatorsMessage(uint256 numberOfValidators, uint256 epochNumber)
67          internal
68          pure
69          returns (bytes32)
70      {
71          return keccak256(abi.encode(numberOfValidators, epochNumber)).toEthSignedMessageHash();
72      }
73
74      /**
75       * @notice Returns the message to be signed for the no restaking module rewards root
76       * @param moduleName is the name of the module
77       * @param root is the root of the no restaking module rewards
78       * @param blockNumber is the block number of the no restaking module rewards
79       * @return the message to be signed
80       */
81      function _getModuleRewardsRootMessage(bytes32 moduleName, bytes32 root, uint256 blockNumber
            )
82          internal
83          pure
84          returns (bytes32)
```

```
85      {
86          return keccak256(abi.encode(moduleName, root, blockNumber)).toEthSignedMessageHash();
87      }
88  }
```

Listing 2.3: PufferPool/src/LibGuardianMessages.sol

Impact Collisions can lead to asset loss and DoS depending on the message structures, functions, and access controls.

Suggestion Add type hashes for different messages to prevent message collisions. Please refer to EIP-712 for more information.

Feedback from Client This will be mitigated by setting related functions restricted to authorized addresses. We will use EIP-712 in the future.

2.1.3 Potential Reward Loss Due to Strict Block Number Validation

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `PufferModule` contract, the function `postRewardRoot()` allows anyone to submit a valid, guardian-signed signature for posting rewards root for a specified block number. The function has a requirement in Line 331 that the block number of each post must be larger than the previous one.

However, there might exist two valid signatures that are not validated. If the one whose block number is larger is validated first, the other one cannot be validated, leading to a loss of rewards.

```
324   function postRewardsRoot(bytes32 root, uint256 blockNumber, bytes[] calldata
          guardianSignatures)
325       external
326       virtual
327       whenNotPaused
328   {
329       PufferModuleStorage storage $ = _getPufferModuleStorage();
330
331       if (blockNumber <= $.lastProofOfRewardsBlockNumber) {
332           revert InvalidBlockNumber(blockNumber);
333       }
334
335       IGuardianModule guardianModule = PUFFER_PROTOCOL.GUARDIAN_MODULE();
336
337       bytes32 signedMessageHash = LibGuardianMessages._getModuleRewardsRootMessage($.moduleName,
              root, blockNumber);
338
339       bool validSignatures = guardianModule.validateGuardiansEOASignatures(guardianSignatures,
              signedMessageHash);
340       if (!validSignatures) {
341           revert Unauthorized();
342       }
```

```
343
344         $.lastProofOfRewardsBlockNumber = blockNumber;
345         $.rewardsRoots[blockNumber] = root;
346         emit RewardsRootPosted(blockNumber, root);
347     }
```

Listing 2.4: PufferPool/src/PufferModule.sol

Impact Rewards can be lost.

Suggestion Ensure the new message is not signed until the previous one is validated.

Feedback from the Project The rewards are not enabled yet. A completely different withdrawal flow will be used in the future.

2.1.4 Potential Inconsistency in the bondAmount Calculation

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `registerValidatorKey()` in the contract `PufferProtocol` is permissionless for users to register as node operators. The registration requires the user to lock a `minimumVtAmount` of `ValidatorTickets`.

In this case, users may purchase the `ValidatorTickets` during the registration process. Note that purchasing `ValidatorTickets` involves transferring Ethers directly to the `PufferVault`, which consequently increases the exchange rate of `pufETH`.

However, the required `bondAmount` of `pufETHs` can be either calculated before (line 212) or after (line 234) the purchase of `ValidatorTickets`, leading to an inconsistent calculation.

```
196     function registerValidatorKey(
197         ValidatorKeyData calldata data,
198         bytes32 moduleName,
199         Permit calldata pufETHPermit,
200         Permit calldata vtPermit
201     ) external payable restricted {
202         ProtocolStorage storage $ = _getPufferProtocolStorage();
203
204         // Revert if the permit amounts are non zero, but the msg.value is also non zero
205         if (vtPermit.amount != 0 && pufETHPermit.amount != 0 && msg.value > 0) {
206             revert InvalidETHAmount();
207         }
208
209         _checkValidatorRegistrationInputs({ $: $, data: data, moduleName: moduleName });
210
211         uint256 validatorBond = data.raveEvidence.length > 0 ? _ENCLAVE_VALIDATOR_BOND :
                _NO_ENCLAVE_VALIDATOR_BOND;
212         uint256 bondAmount = PUFFER_VAULT.convertToShares(validatorBond);
213         uint256 vtPayment = pufETHPermit.amount == 0 ? msg.value - validatorBond : msg.value;
214
215         uint256 receivedVtAmount;
216         // If the VT permit amount is zero, that means that the user is paying for VT with ETH
```

```
217         if (vtPermit.amount == 0) {
218             receivedVtAmount = VALIDATOR_TICKET.purchaseValidatorTicket{ value: vtPayment }(address
                    (this));
219         } else {
220             _callPermit(address(VALIDATOR_TICKET), vtPermit);
221             receivedVtAmount = vtPermit.amount;
222
223             // slither-disable-next-line unchecked-transfer
224             VALIDATOR_TICKET.transferFrom(msg.sender, address(this), receivedVtAmount);
225         }
226
227         if (receivedVtAmount < $.minimumVtAmount) {
228             revert InvalidVTAmount();
229         }
230
231         // If the pufETH permit amount is zero, that means that the user is paying the bond with
                ETH
232         if (pufETHPermit.amount == 0) {
233             // Mint pufETH and store the bond amount
234             bondAmount = PUFFER_VAULT.depositETH{ value: validatorBond }(address(this));
235         } else {
236             _callPermit(address(PUFFER_VAULT), pufETHPermit);
237
238             // slither-disable-next-line unchecked-transfer
239             PUFFER_VAULT.transferFrom(msg.sender, address(this), bondAmount);
240         }
241
242         _storeValidatorInformation({
243             $: $,
244             data: data,
245             pufETHAmount: bondAmount,
246             moduleName: moduleName,
247             vtAmount: receivedVtAmount
248         });
249     }
```

Listing 2.5: PufferPool/src/PufferProtocol.sol

Impact Purchasing `ValidatorTickets` can affect the exchange rate, leading to inconsistent calculation of `bondAmount`.

Suggestion Adjust the order of operations in the code to prevent inconsistencies in the `bondAmount` calculation.

2.1.5 Potential DoS attack due to the depositRootHash verification

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The function `provisionNode()` in the contract `PufferProtocol` validates the `depositRootHash` parameter provided by the user in Line 262. Specifically, in the case of no

usage of enclave, the `depositRootHash` must match the result returned from the external call of `BEACON_DEPOSIT_CONTRACT.get_deposit_root()`. However, the function `get_deposit_root()` in the contract `DepositContract` is dependent on the `deposit_count` variable, which is susceptible to front-run manipulation.

An attacker can front-run a node provisioning transaction and deposit at least 1 ether into the deposit contract to manipulate the expected `depositRootHash`, thereby causing the node provisioning to fail.

```
255   function provisionNode(
256       bytes[] calldata guardianEnclaveSignatures,
257       bytes calldata validatorSignature,
258       bytes32 depositRootHash
259   ) external restricted {
260       // We only use depositRootHash in the no enclave case.
261       // For enclave case, we don't need to check the depositRootHash
262       if (depositRootHash != bytes32(0) && depositRootHash != BEACON_DEPOSIT_CONTRACT.
              get_deposit_root()) {
263           revert InvalidDepositRootHash();
264       }
265
266       ProtocolStorage storage $ = _getPufferProtocolStorage();
267
268       (bytes32 moduleName, uint256 index) = getNextValidatorToProvision();
269
270       // Increment next validator to be provisioned index, panics if there is no validator for
              provisioning
271       $.nextToBeProvisioned[moduleName] = index + 1;
272       unchecked {
273           // Increment module selection index
274           ++$.moduleSelectIndex;
275       }
276
277       _validateSignaturesAndProvisionValidator({
278           $: $,
279           moduleName: moduleName,
280           index: index,
281           guardianEnclaveSignatures: guardianEnclaveSignatures,
282           validatorSignature: validatorSignature
283       });
284
285       // Update Node Operator info
286       address node = $.validators[moduleName][index].node;
287       --$.nodeOperatorInfo[node].pendingValidatorCount;
288       ++$.nodeOperatorInfo[node].activeValidatorCount;
289
290       // Mark the validator as active
291       $.validators[moduleName][index].status = Status.ACTIVE;
292   }
```

Listing 2.6: PufferPool/src/PufferProtocol.sol

```
84   function get_deposit_root() override external view returns (bytes32) {
```

```
85        bytes32 node;
86        uint size = deposit_count;
87        for (uint height = 0; height < DEPOSIT_CONTRACT_TREE_DEPTH; height++) {
88            if ((size & 1) == 1)
89                node = sha256(abi.encodePacked(branch[height], node));
90            else
91                node = sha256(abi.encodePacked(node, zero_hashes[height]));
92            size /= 2;
93        }
94        return sha256(abi.encodePacked(
95            node,
96            to_little_endian_64(uint64(deposit_count)),
97            bytes24(0)
98        ));
99    }
```

Listing 2.7: DepositContract.sol

Impact Node provisioning may fail due to potential front-running.

Suggestion Revise the verification process for the depositRootHash.

Feedback from Client This is a mitigation of front-running setting withdrawal credentials attacks.

2.1.6 Lack of Limits Check in Function `changeMinimumVTAmount()` and `setVTPenalty()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Both function `changeMinimumVTAmount()` and `setVTPenalty()` do not ensure `minimumVtAmount` is larger than `vtPenalty`. If `minimumVtAmount` is smaller than `vtPenalty`, guardians cannot invoke the function `skipProvisioning()`, and the protocol cannot work until a new `vtPenalty` or `minimumVtAmount` is set.

```
734    function _changeMinimumVTAmount(uint256 newMinimumVtAmount) internal {
735        ProtocolStorage storage $ = _getPufferProtocolStorage();
736        emit MinimumVTAmountChanged($.minimumVtAmount, newMinimumVtAmount);
737        $.minimumVtAmount = newMinimumVtAmount;
738    }
```

Listing 2.8: PufferPool/src/PufferProtocol.sol

```
684    function _setVTPenalty(uint256 newPenaltyAmount) internal {
685        ProtocolStorage storage $ = _getPufferProtocolStorage();
686        emit VTPenaltyChanged($.vtPenalty, newPenaltyAmount);
687        $.vtPenalty = newPenaltyAmount;
688    }
```

Listing 2.9: PufferPool/src/PufferProtocol.sol

Impact The `_poolFeeCollected` Temporary protocol DoS.

Suggestion Add the check accordingly to ensure `vtPenalty` is always smaller than `minimumVtAmount`.

2.1.7 Lack of Check on Node Operator Existence Description

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The function `depositValidatorTickets()` allows validator tickets to be deposited to a specified node operator. However, the function does not check whether the node operator has been registered before.

In this case, the deposited tickets may get locked if the non-existing node operator cannot interact with the contract `PufferProtocol`. Specifically, the address cannot withdraw or consume the deposited tickets.

```
140   function depositValidatorTickets(Permit calldata permit, address node) external restricted {
141       if (node == address(0)) {
142           revert InvalidAddress();
143       }
144       // owner: msg.sender is intentional
145       // We only want the owner of the Permit signature to be able to deposit using the signature
146       // For an invalid signature, the permit will revert, but it is wrapped in try/catch,
              meaning the transaction execution
147       // will continue. If the `msg.sender` did a `VALIDATOR_TICKET.approve(spender, amount)`
              before calling this
148       // And the spender is `msg.sender` the Permit call will revert, but the overall transaction
              will succeed
149       try IERC20Permit(address(VALIDATOR_TICKET)).permit({
150           owner: msg.sender,
151           spender: address(this),
152           value: permit.amount,
153           deadline: permit.deadline,
154           v: permit.v,
155           s: permit.s,
156           r: permit.r
157       }) { } catch { }
158
159       // slither-disable-next-line unchecked-transfer
160       VALIDATOR_TICKET.transferFrom(msg.sender, address(this), permit.amount);
161
162       ProtocolStorage storage $ = _getPufferProtocolStorage();
163       $.nodeOperatorInfo[node].vtBalance += SafeCast.toUint96(permit.amount);
164       emit ValidatorTicketsDeposited(node, msg.sender, permit.amount);
165   }
```

Listing 2.10: PufferPool/src/PufferProtocol.sol

Impact The validator tickets may get locked when deposited for a non-existing node operator.

Suggestion Add a node operator existence check in the function `depositValidatorTickets()`.

Feedback from Client Validators might want to pre-deposit their public keys before they enter the entry queue.

2.1.8 Lack of Refund in the `registerValidatorKey()` Function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `registerValidatorKey()` in the contract `PufferProtocol` may convert the deposited Ether into pufETH and purchase the `ValidatorTickets` for users.

When the `vtPermit.amount` is non-zero and `pufETHPermit.amount` is zero, the function transfers the specified `vtPermit.amount` of `ValidatorTickets` from users and deposits a fixed `validatorBond` amount of ETH into the PufferVault contract. However, when the `msg.value` exceeds the required `validatorBond`, the surplus funds (i.e., `msg.value - validatorBond`) are not used for purchasing additional `ValidatorTickets`, resulting in them being locked.

```
196   function registerValidatorKey(
197       ValidatorKeyData calldata data,
198       bytes32 moduleName,
199       Permit calldata pufETHPermit,
200       Permit calldata vtPermit
201    ) external payable restricted {
202       ProtocolStorage storage $ = _getPufferProtocolStorage();
203
204       // Revert if the permit amounts are non zero, but the msg.value is also non zero
205       if (vtPermit.amount != 0 && pufETHPermit.amount != 0 && msg.value > 0) {
206           revert InvalidETHAmount();
207       }
208
209       _checkValidatorRegistrationInputs({ $: $, data: data, moduleName: moduleName });
210
211       uint256 validatorBond = data.raveEvidence.length > 0 ? _ENCLAVE_VALIDATOR_BOND :
              _NO_ENCLAVE_VALIDATOR_BOND;
212       uint256 bondAmount = PUFFER_VAULT.convertToShares(validatorBond);
213       uint256 vtPayment = pufETHPermit.amount == 0 ? msg.value - validatorBond : msg.value;
214
215       uint256 receivedVtAmount;
216       // If the VT permit amount is zero, that means that the user is paying for VT with ETH
217       if (vtPermit.amount == 0) {
218           receivedVtAmount = VALIDATOR_TICKET.purchaseValidatorTicket{ value: vtPayment }(address
                (this));
219       } else {
220           _callPermit(address(VALIDATOR_TICKET), vtPermit);
221           receivedVtAmount = vtPermit.amount;
222
223           // slither-disable-next-line unchecked-transfer
224           VALIDATOR_TICKET.transferFrom(msg.sender, address(this), receivedVtAmount);
225       }
226
227       if (receivedVtAmount < $.minimumVtAmount) {
```

```
228             revert InvalidVTAmount();
229         }
230
231         // If the pufETH permit amount is zero, that means that the user is paying the bond with
                ETH
232         if (pufETHPermit.amount == 0) {
233             // Mint pufETH and store the bond amount
234             bondAmount = PUFFER_VAULT.depositETH{ value: validatorBond }(address(this));
235         } else {
236             _callPermit(address(PUFFER_VAULT), pufETHPermit);
237
238             // slither-disable-next-line unchecked-transfer
239             PUFFER_VAULT.transferFrom(msg.sender, address(this), bondAmount);
240         }
```

Listing 2.11: PufferPool/src/PufferProtocol.sol

Impact The user's surplus funds are locked in the PufferProtocol contract.

Suggestion Implement refund logic for the function `registerValidatorKey()`.

2.2 Additional Recommendation

2.2.1 Typo in `docs/PufferProtocol.md`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In `PufferProtocol.md`, the project describes three scenarios of `withdrawalAmount` in the function `batchHandleWithdrawals()`. The second title should be `32 ETH > withdrawalAmount`.

```
110   2. <span style="color:orange">32 ETH < withdrawalAmount</span>
111
112   `withdrawalAmount` ETH is transferred to the PufferVault and `32 ETH - withdrawalAmount` is
          burned from the Node Operator's bond. It is assumed that the Guardians will eject the
          validator far before inactivity penalties lead to `32 ETH - withdrawalAmount > bond`. The
          Node Operator receives part of their bond back.
113
114   3. <span style="color:red">Validator was slashed</span>
115
116   `withdrawalAmount` is transferred to the `PufferVault` and the **entire** bond will be burned
          regardless of the slashing amount. If there is a major slashing incident, the losses will
          be socialized across all pufETH holders.
```

Listing 2.12: pufferpool/docs/PufferProtocol.md

Suggestion Change `<` operator to `>`.

2.2.2 Redundant Code in the Constructor of `PufferVaultV2`

Status Confirmed

Introduced by `Version 1`

Description `PufferVaultV2` is an upgradable contract, and there are unnecessary storage changes in its constructor (Lines 59 - 63).

```
47  constructor(
48      IStETH stETH,
49      IWETH weth,
50      ILidoWithdrawalQueue lidoWithdrawalQueue,
51      IStrategy stETHStrategy,
52      IEigenLayer eigenStrategyManager,
53      IPufferOracle oracle,
54      IDelegationManager delegationManager
55  ) PufferVault(stETH, lidoWithdrawalQueue, stETHStrategy, eigenStrategyManager) {
56      _WETH = weth;
57      PUFFER_ORACLE = oracle;
58      _DELEGATION_MANAGER = delegationManager;
59      ERC4626Storage storage erc4626Storage = _getERC4626StorageInternal();
60      erc4626Storage._asset = _WETH;
61      _setDailyWithdrawalLimit(100 ether);
62      _updateDailyWithdrawals(0);
63      _setExitFeeBasisPoints(100); // 1%
64      _disableInitializers();
65  }
```

Listing 2.13: pufETH/src/PufferVaultV2.sol

Suggestion Remove the redundant code.

Feedback from the Project This redundant code is for the Echidna fuzz testing.

2.2.3 Add Range Checks in Function `initiateETHWithdrawalsFromLido()`

Status Confirmed

Introduced by `Version 1`

Description According to Lido's documentation [1], the parameter `amounts` of function `requestWithdrawals()` must be between `MIN_STETH_WITHDRAWAL_AMOUNT` and `MAX_STETH_WITHDRAWAL_AMOUNT`. However, this value is not checked, which can lead to a revert in the Lido.

```
251  function initiateETHWithdrawalsFromLido(uint256[] calldata amounts)
252      external
253      virtual
254      override
255      restricted
256      returns (uint256[] memory requestIds)
257  {
258      VaultStorage storage $ = _getPufferVaultStorage();
259
260      uint256 lockedAmount;
261      for (uint256 i = 0; i < amounts.length; ++i) {
262          lockedAmount += amounts[i];
```

[1] https://docs.lido.fi/contracts/withdrawal-queue-erc721/#requestwithdrawals

```
263         }
264         $.lidoLockedETH += lockedAmount;
265
266         SafeERC20.safeIncreaseAllowance(_ST_ETH, address(_LIDO_WITHDRAWAL_QUEUE), lockedAmount);
267         requestIds = _LIDO_WITHDRAWAL_QUEUE.requestWithdrawals(amounts, address(this));
268
269         for (uint256 i = 0; i < requestIds.length; ++i) {
270             $.lidoWithdrawalAmounts.set(requestIds[i], amounts[i]);
271         }
272         emit RequestedWithdrawals(requestIds);
273         return requestIds;
274     }
```

Listing 2.14: pufETH/src/PufferVaultV2.sol

Suggestion Check the value of `amounts` in the function `initiateETHWithdrawalsFromLido()`.

2.2.4 Remove Used `requestIds` in Function `claimWithdrawalsFromLido()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `claimWithdrawalsFromLido()`, the claimed request id is not removed after the project gets the `lidoWithdrawalAmounts` with `requestIds[i]` as the index, which could make `claimWithdrawalsFromLido()` revert if a request id is reused in the future.

```
281     function claimWithdrawalsFromLido(uint256[] calldata requestIds) external virtual override
            restricted {
282         VaultStorage storage $ = _getPufferVaultStorage();
283
284         // ETH balance before the claim
285         uint256 balanceBefore = address(this).balance;
286
287         uint256 expectedWithdrawal = 0;
288
289         for (uint256 i = 0; i < requestIds.length; ++i) {
290             // .get reverts if requestId is not present
291             expectedWithdrawal += $.lidoWithdrawalAmounts.get(requestIds[i]);
292
293             // slither-disable-next-line calls-loop
294             _LIDO_WITHDRAWAL_QUEUE.claimWithdrawal(requestIds[i]);
295         }
296
297         // ETH balance after the claim
298         uint256 balanceAfter = address(this).balance;
299         uint256 actualWithdrawal = balanceAfter - balanceBefore;
300         // Deduct from the locked amount the expected amount
301         $.lidoLockedETH -= expectedWithdrawal;
302
303         emit ClaimedWithdrawals(requestIds);
304         emit LidoWithdrawal(expectedWithdrawal, actualWithdrawal);
305     }
```

Listing 2.15: pufETH/src/PufferVaultV2.sol

Suggestion Remove the `requestIds[i]` mapping after it is used.

2.2.5 Improper Check in the Function `_setEjectionThreshold()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `_setEjectionThreshold`() checks that the `newThreshold` is no larger than 32 ether. However, this check is insufficient as the `newThreshold` can be exactly 32 ether. In this case, a validator can be ejected once it is registered.

```
426   function _setEjectionThreshold(uint256 newThreshold) internal {
427       if (newThreshold > 32 ether) {
428           revert InvalidThreshold(newThreshold);
429       }
430
431       emit EjectionThresholdChanged(_ejectionThreshold, newThreshold);
432       _ejectionThreshold = newThreshold;
433   }
```

Listing 2.16: PufferPool/src/GuardianModule.sol

Suggestion Change `>` operator to `>=`.

2.2.6 Improper Check in the Function `_checkValidatorRegistrationInputs()`

Status Confirmed

Introduced by `Version 1`

Description In the contract `PufferProtocol`, the function `_checkValidatorRegistrationInputs()` verifies that `blsPubKeySet` length matches the threshold instead of the number of guardians. This is inconsistent with the code comment, requiring the number of BLS public key shares to match the guardian number.

```
729   if (data.blsPubKeySet.length != (GUARDIAN_MODULE.getThreshold() * _BLS_PUB_KEY_LENGTH)) {
730       revert InvalidBLSPublicKeySet();
731   }
```

Listing 2.17: PufferPool/src/PufferProtocol.sol

```
29   /**
30    * @notice Thrown when the number of BLS public key shares doesn't match guardians number
31    * @dev Signature "0x8cdea6a6"
32    */
33   error InvalidBLSPublicKeySet();
```

Listing 2.18: PufferPool/src/interface/IPufferProtocol.sol

Suggestion Revise the check to require the `blsPubKeySet` length to match the guardian number.

Feedback from the Project This is by design, and the comments have been corrected.

2.3 Note

2.3.1 Validator Selection is Based on the Module instead of Register Time

Description When guardians provision a validator with funds, the validator selection follows a round-robin style on modules, as shown in the Function `getNextValidatorToProvision()`.

This may lead to an unfair situation. Specifically, a validator will get provision funds after another validator when its module is later in the selection round, even if it is registered first.

```
501   function getNextValidatorToProvision() public view returns (bytes32, uint256) {
502       ProtocolStorage storage $ = _getPufferProtocolStorage();
503
504       uint256 moduleSelectionIndex = $.moduleSelectIndex;
505       uint256 moduleWeightsLength = $.moduleWeights.length;
506       // Do Weights number of rounds
507       uint256 moduleEndIndex = moduleSelectionIndex + moduleWeightsLength;
508
509       // Read from the storage
510       bytes32 moduleName = $.moduleWeights[moduleSelectionIndex % moduleWeightsLength];
511
512       // Iterate through all modules to see if there is a validator ready to be provisioned
513       while (moduleSelectionIndex < moduleEndIndex) {
514           // Read the index for that moduleName
515           uint256 pufferModuleIndex = $.nextToBeProvisioned[moduleName];
516
517           // If we find it, return it
518           if ($.validators[moduleName][pufferModuleIndex].status == Status.PENDING) {
519               return (moduleName, pufferModuleIndex);
520           }
521           unchecked {
522               // If not, try the next module
523               ++moduleSelectionIndex;
524           }
525           moduleName = $.moduleWeights[moduleSelectionIndex % moduleWeightsLength];
526       }
527
528       // No validators found
529       return (bytes32("NO_VALIDATORS"), type(uint256).max);
530   }
```

Listing 2.19: PufferPool/src/PufferProtocol.sol

2.3.2 Access Control is Aligned with Function Annotations

Description Puffer utilizes OpenZepplin's AccessManager, which is governed by `TimeLock`, to manage access controls. During this audit, we assume that the access control is aligned

with function annotations. Specifically, each function with the `restricted` modifier is limited to callers specified by each function annotation, if such an annotation exists.

2.3.3 Potential Centralization Risks

Description The upgrade of the contracts of Puffer is controlled by a multi-signature wallet. If most of the private keys in this wallet are controlled by the same entity, or the private keys are leaked, the vault can be replaced with a malicious contract and cause the loss of users' assets.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS