Security Review Report
NM-0202 PUFFER

NETHERMIND
SECURITY

(Apr 16, 2024)

Contents

1 Executive Summary

This document outlines the security review conducted by Nethermind Security for the Puffer contracts. **Puffer** is a decentralized native liquid restaking protocol (nLRP) built on Eigenlayer. The audit focuses on two sets of contracts: PufferPool and pufETH.

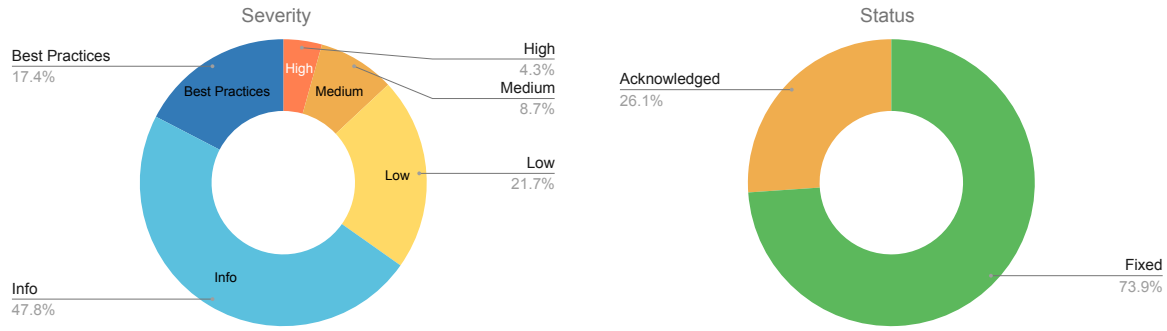
PufferPool is primarily handling interactions with node operators. Its functions include validator registration, node provisioning, rewards distribution, restaking, and validator exiting. These operations are safeguarded by the guardian module, requiring a threshold number of guardian signatures for execution.

On the other hand, **pufETH** is the main point of interaction for stakers. Stakers can deposit ETH into PufferVault and receive yield-bearing pufETH tokens in return. The ETH deposited into PufferVault is used to provision nodes in the Puffer Protocol. Node operators must purchase a validator ticket, after which their validators are provisioned with 32 ETH to begin their validator duties. Due to validator ticket sales, the value of the pufETH token will increase over time.

The audited code consists of 2,351 lines of Solidity. Of these lines, 2,016 corresponds to **PufferPool**. The remaining 335 are connected to **pufETH**, focusing on the V2 upgrade of **PufferVaultV2** and **PufferDepositorV2** contracts. Overall, the audited code is of high quality and shows a strong understanding of Solidity's best practices. The team provided documentation presenting the main use cases and protocol mechanisms. This documentation was supported by a walkthrough of the code and continuous communication with the team.

The audit was performed using: (a) manual analysis of the codebase, (b) simulation of the smart contracts and (c) automation analysis of the smart contracts. **Along this document, we report** 23 points of attention, where one is classified as `High`, two are classified as `Medium`, five are classified as `Low`, and fifteen 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 of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. 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.



(a) distribution of issues according to the severity

(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (1), **Medium** (2), **Low** (5), **Undetermined** (0), **Informational** (11), **Best Practices** (4). **(b) Distribution of status: Fixed** (17), **Acknowledged** (6), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Apr 12, 2024
Final Report	Apr 16, 2024
Methods	Manual Review, Automated analysis
Repository	PufferPool & pufETH
Commit Hash (Initial Audit - PufferPool)	0c60cef0141ef7d04ff8963e98d916ff8075fa2a
Commit Hash (Initial Audit - pufETH)	1b8ec764c17e655dec6b30af54939f88adbb32aa
Final Commit Hash (PufferPool)	2c25db3cc72346b8cc737ddc5b0570ae7477daea
Final Commit Hash (pufETH)	a510fd5924753b77bf0d98795b491bc28729e2e5
Documentation	Protocol docs
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

2.1 PufferPool

	Contract	LoC	Comments	Ratio	Blank	Total
1	LibGuardianMessages.sol	40	41	102.5%	8	89
2	PufferModuleManager.sol	183	72	39.3%	39	294
3	PufferProtocolStorage.sol	11	11	100.0%	3	25
4	Errors.sol	3	9	300.0%	2	14
5	EnclaveVerifier.sol	53	32	60.4%	16	101
6	ValidatorTicket.sol	117	61	52.1%	24	202
7	PufferOracle.sol	28	24	85.7%	8	60
8	ValidatorTicketStorage.sol	14	28	200.0%	3	45
9	PufferOracleV2.sol	66	62	93.9%	17	145
10	PufferModule.sol	238	119	50.0%	51	408
11	RestakingOperator.sol	81	50	61.7%	19	150
12	LibBeaconchainContract.sol	25	15	60.0%	1	41
13	GuardianModule.sol	262	133	50.8%	65	460
14	PufferProtocol.sol	460	205	44.6%	122	787
15	interface/IEnclaveVerifier.sol	20	35	175.0%	8	63
16	interface/IValidatorTicket.sol	17	55	323.5%	14	86
17	interface/IBeaconDepositContract.sol	13	10	76.9%	2	25
18	interface/IRestakingOperator.sol	11	33	300.0%	7	51
19	interface/IWETH.sol	18	4	22.2%	2	24
20	interface/IPufferModule.sol	47	71	151.1%	18	136
21	interface/IPufferModuleManager.sol	79	166	210.1%	29	274
22	interface/IPufferProtocol.sol	85	237	278.8%	53	375
23	interface/IGuardianModule.sol	60	163	271.7%	33	256
24	struct/StoppedValidatorInfo.sol	10	11	110.0%	1	22
25	struct/Validator.sol	9	4	44.4%	2	15
26	struct/NodeInfo.sol	6	4	66.7%	1	11
27	struct/ProtocolStorage.sol	20	52	260.0%	3	75
28	struct/Status.sol	8	4	50.0%	1	13
29	struct/RaveEvidence.sol	6	4	66.7%	1	11
30	struct/ValidatorKeyData.sol	9	4	44.4%	1	14
31	integrations/BalancerRateProvider.sol	17	15	88.2%	5	37
	Total	**2016**	**1734**	**86.0%**	**559**	**4309**

2.2 pufETH

	Contract	LoC	Comments	Ratio	Blank	Total
1	PufferVaultV2.sol	275	174	63.3%	71	520
2	PufferDepositorV2.sol	60	23	38.3%	13	96
	Total	**335**	**197**	**58.8%**	**84**	**616**

3 Summary of Issues

	Finding	Severity	Update
1	Potential front-running `provisionNode(...)` to overwrite withdrawal credential	High	Fixed
2	Potential hash collision between different guardian messages	Medium	Acknowledged
3	The check in setting fees is wrong	Medium	Fixed
4	Incorrect index used when calculating `sharesWithdrawn`	Low	Fixed
5	No check of low-level call execution	Low	Fixed
6	Threshold is not checked when removing guardians	Low	Fixed
7	Unfair module choosing when some modules are empty	Low	Acknowledged
8	`numberOfActiveValidators` variable is not updated when `skipProvisioning(...)` is called	Low	Fixed
9	Changing the daily withdrawal limit resets daily withdrawals	Info	Acknowledged
10	Disabling function `depositToEigenLayer(...)` in PufferVaultV2	Info	Fixed
11	Duplicate code	Info	Fixed
12	Function `addGuardian(...)` does not check if the address to be added as a Guardian is an EOA	Info	Acknowledged
13	Inappropriate names for functions and variables	Info	Fixed
14	Invariant violation in function `setValidatorLimitPerModule(...)`	Info	Fixed
15	Lack of slippage protection when purchasing validator ticket	Info	Acknowledged
16	Missing input validations	Info	Fixed
17	Module is not validated to have a name different than `NO_VALIDATORS`	Info	Fixed
18	Slasher is not implemented in the upgraded version of EigenLayer contracts	Info	Acknowledged
19	Unnecessary usage of `_disableInitializers()` in the constructor of `PufferDepositorV2`	Info	Fixed
20	Incorrect comments about ValidatorTicket fee	Best Practices	Fixed
21	Lack of `_disableInitializers()` in some upgradable contracts	Best Practices	Fixed
22	Unnecessary cast to `uint56` when setting price	Best Practices	Fixed
23	Unnecessary external call to get deposit data root	Best Practices	Fixed

4 System Overview

Puffer is a decentralized native liquid restaking protocol (nLRP) built on Eigenlayer. It makes native restaking on Eigenlayer more accessible, allowing anyone to run an Ethereum Proof of Stake (PoS) validator while supercharging their rewards.

The **Puffer** Protocol comprises two sets of contracts: **pufETH** and **PufferPool**

- **pufETH** is the main point of interaction for stakers. Stakers can deposit ETH and mint the pufETH nLRT via the PufferVault contract, which serves as a redeemable receipt for their restaked ETH. If sufficient exit liquidity is available, stakers can reclaim their ETH from the PufferVault. Over time, the redeemable amount is expected to increase from validator tickets and restaking rewards.
- **PufferPool** is primarily handling interactions with node operators. Its functions include validator registration, node provisioning, rewards distribution, restaking, and validator exiting. These operations are safeguarded by the guardian module, requiring a threshold number of guardian signatures for execution.

Fig. 2 The diagram shows interactions between Puffer contracts and external entities.

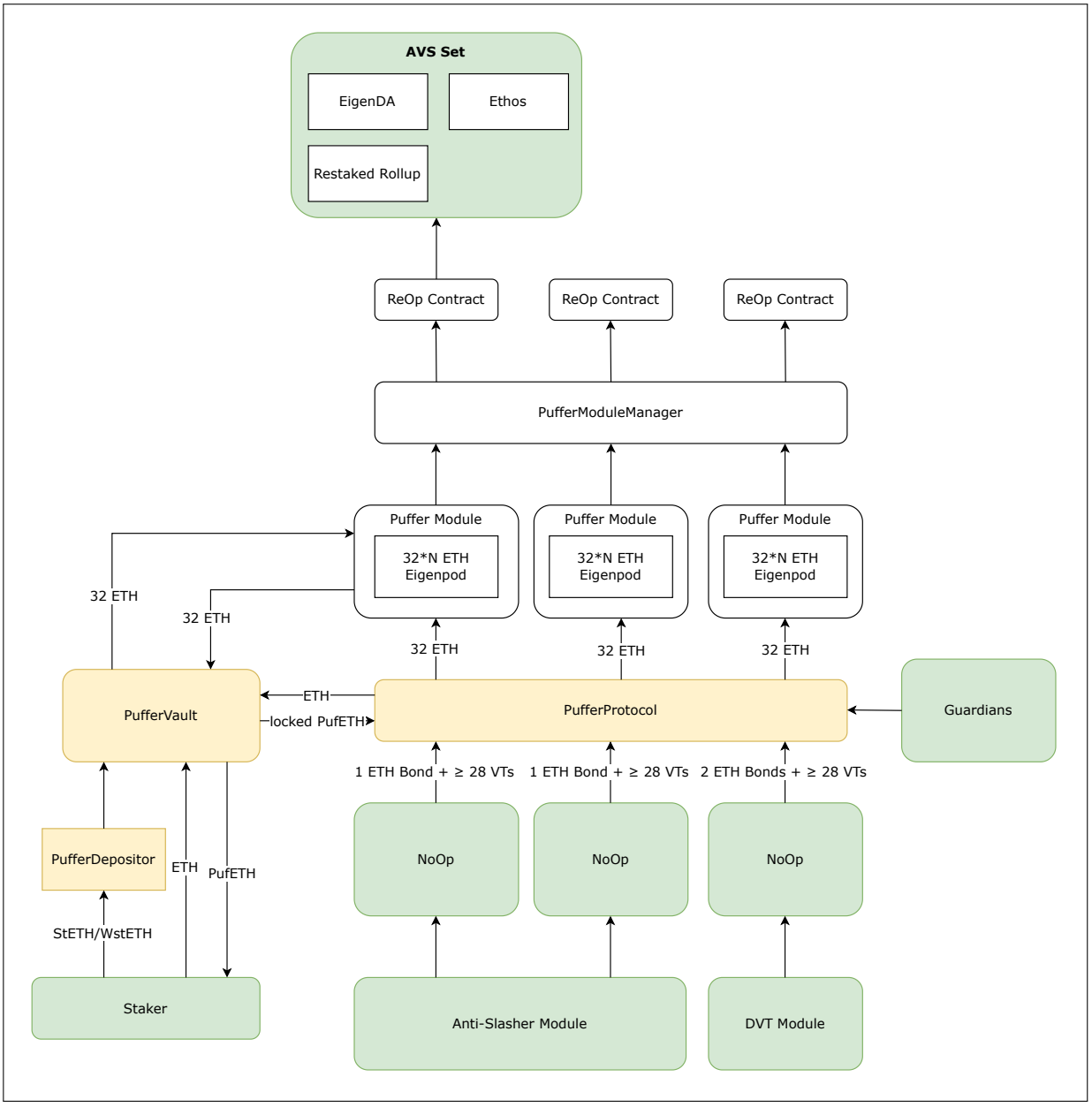


Fig. 2: Puffer Interaction Diagram

4.1 pufETH

pufETH is a native liquid restaking token (nLRT). Users can deposit ETH and mint the pufETH nLRT through the PufferVault contract, serving as a redeemable receipt for their restaked ETH. Current pufETH holders earn LST yield, Puffer points and can participate in DeFi without lockups.

Before Puffer's full mainnet launch, users can deposit stETH into the PufferVault to participate in Puffer's early adopter program. Users without stETH can also participate as the contract supports token swapping to stETH before depositing. In return, depositors receive pufETH, which appreciates as the underlying stETH in the contract accrues.

Upon Puffer's full mainnet launch, stETH will be withdrawn from EigenLayer and converted to ETH via Lido. This process is expected to take approximately 10 days. During this time, depositors can continue to mint pufETH, but the contract will switch to accept ETH & WETH deposits.

After the withdrawal process, the ETH is used to provide decentralized Ethereum validators within the Puffer protocol. This marks a transition from Lido LST rewards to Puffer Protocol rewards. pufETH holders don't need to take any action. However, as the Puffer protocol operates, the value of pufETH is expected to increase faster as it now captures both PoS and restaking rewards.

This audit focused on the mainnet launch upgrade of **pufETH** with two contracts: **PufferVaultV2** and **PufferDepositorV2**.

- **PufferVaultV2** is responsible for holding funds for the Puffer protocol. The initial V1 deployment is an ERC4626 vault with stETH as the underlying asset. The **PufferVaultV2** contract is the next upgrade, changing the underlying asset to WETH and adding functionality to support the Puffer protocol's mainnet deployment. At any point in time, the full backings of **PufferVaultV2** may be a combination of stETH, WETH, and ETH.
- **PufferDepositorV2** facilitates the deposit of stETH and wstETH into the **PufferVaultV2**. Deposit of wstETH will be unwrapped to stETH before depositing to **PufferVaultV2**.

4.2 PufferPool

PufferPool is the core of the Puffer protocol, primarily handling interactions with node operators. The audit scope includes all contracts in this repository:

- **PufferProtocol** is central to managing validator operations, including accounting and custody of validator bonds and VTs. Node operators start by registering a validator using the function `registerValidatorKey(...)`. They can purchase or transfer validator tickets to this contract to start a new validator. Node operators can also specify the **PufferModule** in which they will participate.
- **PufferModuleManager** is a factory contract for creating protocol-owned **PufferModule** and **RestakingOperator** contracts. It also serves as a central contract for coordinating calls to the **PufferModule** and **RestakingOperator** contracts. Interactions from DAO will initiate in the **PufferModuleManager** contract, which will then call the equivalent **PufferModule** and **RestakingOperator**.
- **PufferModule** contract owns an EigenPod and is responsible for managing and interacting with the EigenLayer contracts. As Puffer's permissionless node operators deploy their validators, they set their withdrawal credentials to point to a **PufferModule**'s EigenPod to participate in restaking. **PufferModule** can then delegate their restaked beacon chain ETH to an operator to service AVSs.
- **RestakingOperator** contract handles EigenLayer interactions. This reduces the trust placed in the operators, as the **RestakingOperator** contract controls AVSs selections. All **RestakingOperator** contract functions are restricted to the **PufferModuleManager** contract, giving the DAO control over which AVSs the operator can participate in.
- **GuardianModule** contract is used to manage the Guardians of the Puffer protocol. Guardians play a vital role in Puffer's protocol. They are a collective of respected community members deeply aligned with Ethereum's principles and values. The roles of the Guardians include:
  - Provisioning new validators who registered
  - Skipping malformed validator registrations
  - Ejecting validators whose ETH balance has fallen too low or who have exhausted their Validator Tickets
  - Validator tickets accounting for node operators
  - Handling full withdrawal requests
  - Reporting the number of active Puffer validators
  - Reporting the total number of active Ethereum validators for enforcing the BurstThreshold
- **ValidatorTicket** is used as validator time at a discounted price. To run validators in Puffer, node operators lock VT ERC20s in the PufferProtocol contract. Each validator consumes 1 VT per day they are active. Notably, VTs are consumed only after the validator has been activated on the beacon chain, meaning their tickets do not expire while in the queueing process.
- **PufferOracleV2** contract is used to set important variables for the protocol's accounting. These variables include the number of active Puffer validators, the total number of validators on Ethereum, and the validator ticket price.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind 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 an attacker will uncover and exploit the finding. 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 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 formally pass to the client;

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 [High] Potential front-running `provisionNode(...)` to overwrite withdrawal credential

File(s): `PufferProtocol.sol`

Description: After node operators register a validator, guardians verify the validator's setup and sign a message. This allows anyone to call the `provisionNode(...)` function, provide the signatures, and activate the validator.

The vulnerability stems from the fact that Beacon Chain allows multiple deposits for a single validator public key. It does not check that all deposits have the same withdrawal credentials and assumes the withdrawal credential is in the first valid deposit.

In the Puffer Protocol, the `provisionNode(...)` function stakes 32 ETH for a validator through the EigenPod. This function sets the withdrawal credential to the EigenPod contract. However, if the validator front-runs this `provisionNode(...)` transaction and deposits directly to the Beacon Deposit Contract, they can redirect the withdrawal credential to an address they control instead of the EigenPod.

Recommendation(s): Consider adding the deposit root of the Beacon Deposit Contract to the guardian's signature. In the function `provisionNode(...)`, ensure that the current deposit root matches the provided one.

Status: Fixed

Update from the client: `provisionNode` is now permissioned, and for the no enclave case, we will use the deposit root hash

https://github.com/PufferFinance/PufferPool/pull/224/commits/f8f4ffcecc643650ec79ea0dddf552d5bbfa93e5

Update from Nethermind: The issue is fixed due to the restriction of the function calling to Paymaster. However, the proper fix should include `depositRootHash` in the data for the guardians' signature.

The current design makes a protocol more centralised. The protocol relied initially on N guardians, but now it depends on one Paymaster entity.

Update from the client: The exclusion of `depositRootHash` from the guardian' signature is intentional to shorten the time from setting the deposit root hash and the transaction execution. We will extend the access to this function to the guardians other than the paymaster to prevent centralization.

6.2 [Medium] Potential hash collision between different guardian messages

File(s): `LibGuardianMessages.sol`

Description: The data lengths being hashed in the functions `_getSkipProvisioningMessage(...)` and `_getSetNumberOfValidatorsMessage(...)` are both 64 bytes. This could potentially lead to hash collisions between these two guardian messages.

```
1  function _getSkipProvisioningMessage(bytes32 moduleName, uint256 index) internal pure returns (bytes32) {
2      // All guardians use the same nonce
3      return keccak256(abi.encode(moduleName, index)).toEthSignedMessageHash();
4  }
5
6  function _getSetNumberOfValidatorsMessage(uint256 numberOfValidators, uint256 epochNumber)
7      internal
8      pure
9      returns (bytes32)
10 {
11     return keccak256(abi.encode(numberOfValidators, epochNumber)).toEthSignedMessageHash();
12 }
```

As a result, when a guardian signs a message, the same signature can potentially be used for both skip provisioning and setting the number of validators.

Recommendation(s): Adhere to EIP-712 to avoid hash collision between different types of messages.

Status: Acknowledged

Update from the client: Mitigated by changing the functions to restricted to authorized addresses (paymaster). In the future upgrades, we will update all signed message formats to adhere with to EIP-712.

Making the related functionality `setTotalNumberOfValidators` permissioned.

However, this is highly unlikely to happen given the type of the values encoded are different:

- `abi.encode(moduleName, index)` → moduleName: bytes32, index: uint256;
- `abi.encode(numberOfValidators, epochNumber)` → numberOfValidators: uint256;
- https://github.com/PufferFinance/PufferPool/commit/38d8fce86aff4e50cff26bbe994af72584effafd;

6.3 [Medium] The check in setting fees is wrong

File(s): ValidatorTicket.sol

Description: DAO address can set fees for purchasing validator tickets. There are two categories of fees: Protocol fee and Guardian's fee. The first fee is sent to the protocol's treasury, and the second one to the contract where guardians are managed.

The maximum fee should be 10%, as stated in the following comment:

```
function _setProtocolFeeRate(uint256 newProtocolFeeRate) internal virtual {
  ValidatorTicket storage $ = _getValidatorTicketStorage();
  // Treasury fee can not be bigger than 10%
...
}
```

However, when setting new fees for both protocol and guardians (`_setProtocolFeeRate(...)` and `_setGuardiansFeeRate(...)`), a new fee is not compared with the maximum value at all. In reality, a current fee is compared with the maximum value.

```
function _setProtocolFeeRate(uint256 newProtocolFeeRate) internal virtual {
  ValidatorTicket storage $ = _getValidatorTicketStorage();
   // Treasury fee can not be bigger than 10%
   if ($.protocolFeeRate > (1000)) { // @audit wrong comparison
     revert InvalidData();
      }
     ...
}

function _setGuardiansFeeRate(uint256 newGuardiansFeeRate) internal virtual {
  ValidatorTicket storage $ = _getValidatorTicketStorage();
   // Treasury fee can not be bigger than 10%
   if ($.protocolFeeRate > (1000)) { // @audit wrong comparison
     revert InvalidData();
   }
   ...
}
```

As a result, a fee bigger than 10% is possible. Furthermore, once a fee larger than 10% is set in the current implementation, it will not be possible to change it.

Recommendation(s): Replace the current fee rate in the conditional statement with the new one.

Status: Fixed

Update from the client: Fixed the natspec and using the correct values in the setter functions

- https://github.com/PufferFinance/PufferPool/commit/635a19712a4779072824bdd22e1bb9e026c1d5c5;

6.4 [Low] Incorrect index used when calculating `sharesWithdrawn`

File(s): PufferModuleManager.sol

Description: The function `callCompleteQueuedWithdrawals(...)` calculates the total number of shares withdrawn by performing a nested loop through all the withdrawal shares and summing them up. However, the index used to get the share value is incorrect, resulting in an incorrect calculation.

```
function callCompleteQueuedWithdrawals(
    ...
) external virtual restricted {
    ...
    uint256 sharesWithdrawn;

    for (uint256 i = 0; i < withdrawals.length; i++) {
        for (uint256 j = 0; j < withdrawals[i].shares.length; j++) {
            sharesWithdrawn += withdrawals[i].shares[i]; // @audit Should use withdrawals[i].shares[j] instead
        }
    }
    ...
}
```

Recommendation(s): Use `withdrawals[i].shares[j]` instead of `withdrawals[i].shares[i]` when calculating `sharesWithdrawn`.

Status: Fixed

Update from the client: https://github.com/PufferFinance/PufferPool/commit/065fa2b1906970d615af68a72dda44a5f4d7463e

6.5 [Low] No check of low-level call execution

File(s): PufferProtocol.sol

Description: During the execution of `batchHandleWithdrawals(...)`, a `transferAmount` of ETH is sent from the puffer module to `PufferVault`:

```
IPufferModule(validatorInfos[i].module).call(address(PUFFER_VAULT), transferAmount, "");
```

The line above executes a high-level function `call(...)`. If investigate this function further:

```
function call(address to, uint256 amount, bytes calldata data) external onlyPufferProtocol returns (bool success, bytes
↪  memory) {
    return to.call{ value: amount }(data);
}
```

It executes a low-level `call`, which returns a boolean value indicating its successful execution. However, this boolean value is never checked, which can create issues if not enough ETH balance is present in a module.

Recommendation(s): Consider checking the result of a low-level call.

Status: Fixed

Update from the client: Fixed by checking the return success and revert if fail.

- https://github.com/PufferFinance/PufferPool/commit/42749c7660f7c1bdb9b921148eadb29cdebd8dfd;

6.6 [Low] Threshold is not checked when removing guardians

File(s): GuardianModule.sol

Description: When `removeGuardian(...)` is called, the guardian count may drop below the threshold, so the number of signers would not be enough to create a valid signature, potentially leaving the protocol in an inoperative state until either a new guardian is added or the threshold is updated.

```
    function removeGuardian(address guardian) external restricted {
        splitGuardianFunds();
        (bool success) = _guardians.remove(guardian);
        if (success) {
            emit GuardianRemoved(guardian);
        }
    }
```

Recommendation(s): Consider validating the threshold to ensure that after removing a guardian, the remaining number of guardians is at least equal to the threshold.

Status: Fixed

Update from the client: Fixed by reverting if the number of guardians will go bellow the threshold

- Commit: https://github.com/PufferFinance/PufferPool/commit/34162f96cec7cbf7b039149fb89fc468f3a394da;

6.7 [Low] Unfair module choosing when some modules are empty

File(s): PufferProtocol.sol

Description: The function `provisionNode(...)` calls `getNextValidatorToProvision(...)` to identify the next validator for provisioning. This function loops from the current `moduleSelectIndex` to find the nearest non-empty module.

```
while (moduleSelectionIndex < moduleEndIndex) {
    // Read the index for that moduleName
    uint256 pufferModuleIndex = $.nextToBeProvisioned[moduleName];

    // If we find it, return it
    if ($.validators[moduleName][pufferModuleIndex].status == Status.PENDING) {
        return (moduleName, pufferModuleIndex);
    }
    unchecked {
        // If not, try the next module
        ++moduleSelectionIndex;
    }
    moduleName = $.moduleWeights[moduleSelectionIndex % moduleWeightsLength];
}
```

However, after this, the `moduleSelectIndex` variable only increases by `1`. This does not align with the logic of `getNextValidatorToProvision(...)` and allows a module to be selected more than once in a row.

```
// @audit Skip empty module
(bytes32 moduleName, uint256 index) = getNextValidatorToProvision();
...
// @audit Not match the logic in getNextValidatorToProvision()
unchecked {
    // Increment module selection index
    ++$.moduleSelectIndex;
}
```

Consider this scenario:

1. Let's say we have 4 modules `[mod_1, mod_2, mod_3, mod_4]` and the number of validators in each module are `[0, 0, 3, 3]` (the first and second modules do not have any validator);
2. Assume `moduleSelectIndex = 0`, the `getNextValidatorToProvision(...)` function will choose `mod_3` for provision. After that `moduleSelectIndex = 1` is updated;
3. The next call to `provisionNode(...)`, the function `getNextValidatorToProvision(...)` will still choose `mod_3` for provision. After that `moduleSelectIndex = 2` is updated;
4. Again, the third time function `provisionNode(...)` is called, the function `getNextValidatorToProvision(...)` will still choose `mod_3` for provision;

As demonstrated, validators in `mod_3` are chosen three times in a row.

Recommendation(s): Consider updating `moduleSelectIndex` according to the chosen index after the function `getNextValidatorToProvision(...)`.

Status: Acknowledged

Update from the client: Given that in addition to the order, each module has a weight associated with it, the weights should be set accordingly `setModuleWeights` to prevent the discussed issue.

Update:

- In the warm phase, we'll have a very even distribution in the modules, specially as the validators have control over which module to register their keys to;
- In the case we add new modules that are more popular, (e.g. ABCD are empty and we introduce E) we may use `setModuleWeights([E])` to save the gas costs and prioritizes E module selection;
- Also in the scenario when mod1 is empty, a validator can go to the empty mod to get provisions faster without any queue;
- In addition to that, the DAO can set `setValidatorLimitPerModule(bytes32 moduleName, uint128 limit)` to prevent new registrations on any module that is over populated;

6.8 [Low] `numberOfActiveValidators` variable is not updated when `skipProvisioning(...)` is called

File(s): `PufferProtocol.sol`

Description: During the validator's registration, the variable `numberOfActiveValidators` is increased by one for the module the validator registers. This variable contains a number of active validators the module manages.

But, to make a validator active, it is necessary to call the `provisionNode(...)` function with a signed message from guardians. However, guardians can also skip provisioning to skip the pending validator. This is done in the function `skipProvisioning(...)`.

During the execution of `skipProvisioning(...)`, the `numberOfActiveValidators` is not decreased, leading to inconsistency between the number of active validators and the number stored in that variable.

Recommendation(s): Decrease `numberOfActiveValidators` during a calll of `skipProvisioning(...)`.

Status: Fixed

Update from the client: Added the logic to decrease the number of active validators in skip provisioning.

- https://github.com/PufferFinance/PufferPool/commit/0aba383a3ae14424b2a732727cd94a4d4ede389c;

6.9 [Info] Changing the daily withdrawal limit resets daily withdrawals

File(s): `PufferVaultV2.sol`

Description: In PufferVaultV2, a daily withdrawal limit is set by the DAO. Users cannot exceed this limit within a day. The daily withdrawn amount will be reset when a new day begins.

When the DAO uses `setDailyWithdrawalLimit(...)`, it also resets daily withdrawals by invoking `_resetDailyWithdrawals(...)`. This action resets the withdrawn amount even before the day ends.

Consider the scenario

1. The current withdrawal limit is 10 ETH and the assets withdrawn today total 5 ETH;
2. The DAO then uses `setDailyWithdrawalLimit(...)` to lower the withdrawal limit to 5 ETH;
3. Users should not be able to withdraw more assets since the new limit (5 ETH) has been reached. However, because `setDailyWithdrawalLimit(...)` also resets the daily withdrawal amount, users can withdraw an additional 5 ETH;

Recommendation(s): Consider avoiding the use of `_resetDailyWithdrawals(...)` during `setDailyWithdrawalLimit(...)`.

Status: Acknowledged

Update from the client: This is the expected behaviour. The recommendation is similar to the the original implementation, however after the priori audit, the daily limit is restarted after the update.

6.10 [Info] Disabling function `depositToEigenLayer(...)` in PufferVaultV2

File(s): `PufferVaultV2.sol`

Description: In the new version of PufferVault, the protocol is transitioning to native staking. The team plans to convert all stETH to ETH for node provisioning within the Puffer protocol. Therefore, the function `depositToEigenLayer(...)` is no longer necessary as it deposits stETH to EigenLayer rather than converting them to ETH.

Recommendation(s): Disable the function `depositToEigenLayer(...)` by overriding it in PufferVaultV2.

Status: Fixed

Update from the client: Override the function to disable the code

- https://github.com/PufferFinance/pufETH/commit/5db7863db529e007a43bacd88aa7809332027fae;

6.11 [Info] Duplicate code

File(s): GuardianModule.sol, PufferProtocol.sol

Description: Certain functions in the codebase contain duplicated code.

- In the `constructor`, the first `if` condition is repeated:;

```
constructor(IEnclaveVerifier verifier, address[] memory guardians, uint256 threshold, address pufferAuthority)
    payable
    AccessManaged(pufferAuthority)
{
    // @audit - This `if` is repeated twice
    if (address(verifier) == address(0)) {
        revert InvalidAddress();
    }
    if (address(verifier) == address(0)) {
        revert InvalidAddress();
    }
    ...
}
```

- In `batchHandleWithdrawals(...)`, the function `_getVTBurnAmount(...)` is called three times with the same input parameters to calculate the same value for each validator;

```
burnAmounts.vt += _getVTBurnAmount(validatorInfos[I]);
...
emit ValidatorExited({
    vtBurnAmount: _getVTBurnAmount(validatorInfos[I])
});
...
$.nodeOperatorInfo[validator.node].vtBalance -= SafeCast.toUint96(_getVTBurnAmount(validatorInfos[i]));
```

Recommendation(s): Refactoring is suggested to eliminate duplicate code.

Status: Fixed

Update from the client: Removed the duplicate code

- https://github.com/PufferFinance/PufferPool/commit/0c82677358c09a3eff94ed3033b0793d5b09815a;

6.12 [Info] Function `addGuardian(...)` does not check if the address to be added as a Guardian is an EOA

File(s): GuardianModule.sol

Description: The Guardians must be EOA but the `addGuardian(...)` function does not validate if the address to be added is a contract address or zero address, before adding it as a Guardian.

```
function addGuardian(address newGuardian) external restricted {
        splitGuardianFunds();
        _addGuardian(newGuardian);
}
```

This could be an issue in a scenario where the contract added as Guardian is unable or its administrator is unwilling to perform its duties as a Guardian. Additionally, it could receive funds that would potentially be locked into the contract forever.

Recommendation(s): Consider implementing checks to confirm the guardian address is an EOA. For instance, require the guardian to sign a message and validate the signature during the addition process.

Status: Acknowledged

Update from the client: Given that the guardians are permissioned, we will add this check to the off-chain components to make sure the addresses are EOA before adding them, and if we ever want to support multisig/smartcontract guardians we can check accordingly.

6.13 [Info] Inappropriate names for functions and variables

File(s): ProtocolStorage.sol, PufferModule.sol

Description: Some functions and variables have potentially misleading names.

- The parameter name `numberOfActiveValidators` in the `ModuleLimit` struct is confusing. It only increments in the `_storeValidatorInformation(...)` function of the `PufferProtocol` contract when `registerValidatorKey(...)` is called. However, registering a validator only adds a pending validator to the module, rather than activating a validator. Thus, this parameter should be named `numberOfValidators` or `numberOfRegisteredValidators`. When renaming the variable, it's essential to ensure that the following event tied to this variable is also updated accordingly;

```
event NumberOfActiveValidatorsChanged(bytes32 indexed moduleName, uint256 newNumberOfActiveValidators);
```

- The function `_getPufferProtocolStorage(...)` in `PufferModule` would be more accurately named `_getPufferModuleStorage(...)`;

Recommendation(s): Consider renaming these functions and variables to the suggested names.

Status: Fixed

Update from the client: Renamed to reflect accurate functionality

- https://github.com/PufferFinance/PufferPool/commit/0c82677358c09a3eff94ed3033b0793d5b09815a;
- https://github.com/PufferFinance/PufferPool/pull/224/commits/ec54e2ade89002c957e88ec8c2f8fa0880cc0957;
- https://github.com/PufferFinance/PufferPool/pull/224/commits/1cfa10b825978dcb7655f7eeae2c82fb2fae12ec;

6.14 [Info] Invariant violation in function `setValidatorLimitPerModule(...)`

File(s): PufferProtocol.sol

Description: In the function `setValidatorLimitPerModule(...)` that calls the internal function `_setValidatorLimitPerModule(...)`, the maximum allowed limit of validators per module is set without verifying that the new limit is not less than the current number of validators. This could break the internal coherence of the system.

```
function _setValidatorLimitPerModule(bytes32 moduleName, uint128 limit) internal {
        ProtocolStorage storage $ = _getPufferProtocolStorage();
        emit ValidatorLimitPerModuleChanged($.moduleLimits[moduleName].allowedLimit, limit);
        $.moduleLimits[moduleName].allowedLimit = limit;
}
```

Recommendation(s): Consider adding a validation before setting the new limit, to prevent it from being set to a value lower than the current number of validators.

Status: Fixed

Update from the client: Added the check to revert if limit is reached

- https://github.com/PufferFinance/PufferPool/commit/0c82677358c09a3eff94ed3033b0793d5b09815a;

6.15 [Info] Lack of slippage protection when purchasing validator ticket

File(s): ValidatorTicket.sol

Description: The price of the validator ticket can be changed in `PUFFER_ORACLE`. However, the function `purchaseValidatorTicket(...)` does not have slippage protection, potentially leading to purchases at unfavorable prices.

```
function purchaseValidatorTicket(address recipient) // @audit No slippage protection
    ...
{
    ValidatorTicket storage $ = _getValidatorTicketStorage();

    uint256 mintPrice = PUFFER_ORACLE.getValidatorTicketPrice();
    ...
}
```

Recommendation(s): Consider allowing the caller to specify a minimum number of validator tickets he/she wants to receive for this purchase.

Status: Acknowledged

Update from the client: Acknowledged

6.16 [Info] Missing input validations

File(s): `PufferProtocol.sol`, `EnclaveVerifier.sol`

Description: Several parts of the code do not implement proper input validations. It is good practice to perform input validation to mitigate human errors and other issues.

- There is no validation in the `PufferProtocol` constructor for type address and contract parameters before setting their value. This parameter could be set to an External Owned Account (EOA) address or a zero address. The impact is mitigated by the fact that the execution corresponds to the deployer role. Consider adding a condition to check the input value of these address parameters;
- In the `initialize(...)` function of the `PufferProtocol` contract, `accessManager` is not validated. It can be set to a zero address;
- `node` address in the `depositValidatorTickets(...)` function can be any address, even zero. In this case, the `vtBalance` can be updated to an invalid node address;
- In the constructor of the `EnclaveVerifier` contract, `accessManager` isn't checked for a zero address and `freshnessBlocks` isn't checked for a valid number;
- In the `changeMinimumVTAmount(...)` function, there are no checks on the value of the `newMinimumVTAmount` parameter, which could allow it to be set by mistake to zero or a very low value that could impact the behaviour of the protocol, for example, allowing the registration of validators without validator tickets. The impact is mitigated by the fact that the function must be called only by the guardians of the DAO. Consider adding a validation on the minimum value to set. Additionally, consider adding validation to ensure the new value differs from the old one so it is not written to storage unless necessary;
- In the `_setVTPenalty(...)` function, there is no upper bound for the `newPenaltyAmount`. It should be checked to be less than the minimum VT amount;

Recommendation(s): Consider validating the inputs according to the above recommendations.

Status: Fixed

Update from the client:

Not 100% resolved because of the compiler issues. https://github.com/PufferFinance/PufferPool/pull/224/commits/ec54e2ade89002c957e88ec8c2f8fa088

6.17 [Info] Module is not validated to have a name different than `NO_VALIDATORS`

File(s): `PufferProtocol.sol`

Description: The name `NO_VALIDATORS` is specifically reserved to return when no available validator is found in the function `getNextValidatorToProvision(...)`. However, when a new module is created, there is no verification to ensure its name isn't `NO_VALIDATORS`.

```
1 // No validators found
2 return (bytes32("NO_VALIDATORS"), type(uint256).max);
```

Recommendation(s): Consider adding a check to the function `_createPufferModule(...)` to ensure the `moduleName` isn't `NO_VALIDATORS`.

Status: Fixed

Update from the client:

https://github.com/PufferFinance/PufferPool/pull/224/commits/ec54e2ade89002c957e88ec8c2f8fa0880cc0957

6.18 [Info] Slasher is not implemented in the upgraded version of EigenLayer contracts

File(s): `RestakingOperator.sol`

Description: The Puffer protocol strongly depends on the external protocol EigenLayer. Therefore, it is necessary to monitor all changes within EigenLayer's codebase. The first version of EigenLayer contracts is currently deployed on the Ethereum mainnet. However, a new upgraded version has already been deployed on the testnet, highlighting the coming upgrade.

A Puffer protocol already uses the functionality of the upgraded contracts. But there are still some notes to consider:

- The upgraded version of EigenLayer contracts does not implement **Slasher**;

- From EigenLayer documentation: _ The Slasher contract is under active development and its interface expected to change. We recommend writing slashing logic without integrating with the Slasher at this point in time. Although the Slasher is deployed, it will remain completely paused/unusable during M2. No contracts interact with it, and its design is not finalized. _ [1]

- `PufferVault.sol`, inherited by `PufferVaultV2.sol`, uses functionality from the currently deployed version and will not work after the upgrade;

Recommendation(s): Monitor all changes of EigenLayer contracts and upgrade Puffer contracts if necessary.

Status: Acknowledged

Update from the client: Acknowledged

6.19 [Info] Unnecessary usage of `_disableInitializers()` in the constructor of `PufferDepositorV2`

File(s): `PufferDepositorV2`

Description: The `_disableInitializers()` method is invoked within the constructor of the `PufferDepositorV2` contract, despite the contract lacking any `initialize(...)` functions.

```
  constructor(PufferVaultV2 pufferVault, IStETH stETH) payable {
        PUFFER_VAULT = pufferVault;
        _ST_ETH = stETH;
        _disableInitializers();
    }
```

Recommendation(s): Consider removing the `_disableInitializers()` call from the constructor, or alternatively, consider adding an initialize function if necessary.

Status: Fixed

Update from the client: Removed unnecessary _disableInitializers

- https://github.com/PufferFinance/pufETH/commit/a510fd5924753b77bf0d98795b491bc28729e2e5;

6.20 [Best Practices] Incorrect comments about ValidatorTicket fee

File(s): `ValidatorTicket.sol`

Description: The `_BASIS_POINT_SCALE` for the ValidatorTicket fee is `1e4`, but the comment incorrectly refers to `100% = 1e18`.

```
uint256 private constant _BASIS_POINT_SCALE = 1e4;
...
toSend = amount.mulDiv(rate, _BASIS_POINT_SCALE, Math.Rounding.Ceil);


struct ValidatorTicket {
    /**
     * @dev Protocol fee rate, can be updated by governance (1e20 = 100%, 1e18 = 1%)
     * Slot 0
     */
    uint128 protocolFeeRate;
    /**
     * @dev Guardians fee rate, can be updated by governance (1e20 = 100%, 1e18 = 1%)
     * Slot 0
     */
    uint128 guardiansFeeRate;
}
```

Recommendation(s): Correct the comment about the ValidatorTicket fee.

Status: Fixed

Update from the client: Fixed the comments

- https://github.com/PufferFinance/PufferPool/commit/635a19712a4779072824bdd22e1bb9e026c1d5c5;

6.21 [Best Practices] Lack of `_disableInitializers()` in some upgradable contracts

File(s): `RestakingOperator.sol`, `PufferModuleManager.sol`

Description: The `RestakingOperator` and `PufferModuleManager` contracts are designed to be upgradable and utilize the OpenZeppelin `Initializable` and `UUPSUpgradeable` APIs. However, they failed to lock the implementation contract after deployment. This oversight allows for the possibility that an attacker could frontrun the `initialize` function, potentially seizing control of the implementation contract.

Recommendation(s): In the context of upgradable contracts, OpenZeppelin advises calling the specific function within the constructor as a best practice. It's recommended to include a call to `_disableInitializers()` in the constructors of both contracts.

Status: Fixed

Update from the client: Added _disableInitializer() to both contracts.

- https://github.com/PufferFinance/PufferPool/commit/0c82677358c09a3eff94ed3033b0793d5b09815a;

6.22 [Best Practices] Unnecessary cast to `uint56` when setting price

File(s): `PufferOracleV2.sol`

Description: The data type of `_validatorTicketPrice` is `uint256`. Nevertheless, the value is always cast to `uint56` before it's set. This is unnecessary and may result in slightly higher gas consumption.

```
uint256 internal _validatorTicketPrice;

_setMintPrice(uint56(0.01 ether));
```

Recommendation(s): Consider eliminating the value casting when setting the price.

Status: Fixed

Update from the client: Leftover from old code, fixed by removing the unnecessary casting

- https://github.com/PufferFinance/PufferPool/commit/0c82677358c09a3eff94ed3033b0793d5b09815a;
- https://github.com/PufferFinance/PufferPool/pull/224/commits/ec54e2ade89002c957e88ec8c2f8fa0880cc0957;

6.23 [Best Practices] Unnecessary external call to get deposit data root

File(s): `PufferProtocol.sol`

Description: The function `_validateSignaturesAndProvisionValidator(...)` makes an external call to a function of the PufferProtocol contract to get the deposit data root. However, this is unnecessary and results in wasted gas.

```
bytes32 depositDataRoot = this.getDepositDataRoot({ // @audit No need external call
    pubKey: validatorPubKey,
    signature: validatorSignature,
    withdrawalCredentials: withdrawalCredentials
});
```

Recommendation(s): Consider using `LibBeaconchainContract` directly to get the deposit data root.

Status: Fixed

Update from the client: Using internal call to `LibBeaconchainContract.getDepositDataRoot` instead of an external call.

https://github.com/PufferFinance/PufferPool/commit/f8f4ffcecc643650ec79ea0dddf552d5bbfa93e5diff-24ef907362604f28b204d5236797e5ba1f0e987e7d

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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 Puffer documentation

The `Puffer` team has provided comprehensive documentation about their protocol. This includes their Protocol docs, comments throughout the codebase and /docs folder in the repository. These documents detail the overall architecture of `Puffer` contracts and each component. Additionally, the Puffer team was available to address any questions or concerns from the Nethermind auditors.

8 Test Suite Evaluation

8.1 Compilation Output

8.1.1 PufferPool

```
> forge compile
[] Compiling...
[] Compiling 215 files with 0.8.24
[] Solc 0.8.24 finished in 42.87s
Compiler run successful with warnings
```

8.1.2 pufETH

```
> forge compile
[] Compiling...
[] Compiling 112 files with 0.8.24
[] Solc 0.8.24 finished in 17.45s
Compiler run successful with warnings:
```

8.2 Tests Output

8.2.1 PufferPool

```
> forge test -vvv --match-path './test/unit/*'

Ran 9 tests for test/unit/ValidatorTicket.t.sol:ValidatorTicketTest
[PASS] test_change_protocol_fee_rate() (gas: 40025)
[PASS] test_funds_splitting() (gas: 166532)
[PASS] test_non_whole_number_purchase() (gas: 162264)
[PASS] test_overflow_protocol_fee_rate() (gas: 7859)
[PASS] test_set_guardians_fee_rate() (gas: 41442)
[PASS] test_setup() (gas: 27672)
[PASS] test_split_funds_no_protocol_fee_rate() (gas: 148916)
[PASS] test_zero_protocol_fee_rate() (gas: 38996)
[PASS] test_zero_vt_purchase() (gas: 56968)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 43.84ms (3.08ms CPU time)

Ran 7 tests for test/unit/EnclaveVerifier.t.sol:EnclaveVerifierTest
[PASS] testAddLeafX509() (gas: 193090)
[PASS] testRaveEvidence1() (gas: 899342)
[PASS] testRaveEvidence2() (gas: 899297)
[PASS] testRaveEvidence3() (gas: 899317)
[PASS] testRemoveLeafX509() (gas: 201121)
[PASS] testSetup() (gas: 89204)
[PASS] testVerifyingStaleEvidence() (gas: 737691)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 50.66ms (15.66ms CPU time)

Ran 1 test for test/unit/LibBeaconchainContract.t.sol:LibBeaconChainTest
[PASS] testUpdatedVersion(bytes32) (runs: 256, : 23943, ~: 23943)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 62.65ms (58.66ms CPU time)

Ran 5 tests for test/unit/PufferOracle.t.sol:PufferOracleTest
[PASS] test_mint_price_is_restricted() (gas: 25464)
[PASS] test_set_mint_price_0_reverts() (gas: 25498)
[PASS] test_set_mint_price_exceeds_maximum_reverts(uint256) (runs: 256, : 29912, ~: 29859)
[PASS] test_set_mint_price_succeeds(uint256) (runs: 256, : 39184, ~: 38886)
[PASS] test_setup() (gas: 9243)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 148.04ms (139.10ms CPU time)

Ran 14 tests for test/unit/GuardianModule.t.sol:GuardianModuleTest
[PASS] test_addGuardian(address) (runs: 256, : 133993, ~: 133993)
[PASS] test_rave() (gas: 128603074)
[PASS] test_removeGuardian(address) (runs: 256, : 114104, ~: 114089)
[PASS] test_rotateGuardianKey_from_non_guardian_reverts() (gas: 12139)
[PASS] test_rotateGuardianKey_to_invalid_pubKey_everts() (gas: 14671)
```

```
[PASS] test_rotateGuardianKey_with_invalid_rave_reverts() (gas: 841680)
[PASS] test_set_threshold() (gas: 35354)
[PASS] test_set_threshold_reverts() (gas: 27715)
[PASS] test_set_threshold_to_0_reverts() (gas: 28213)
[PASS] test_set_threshold_to_50_reverts() (gas: 28138)
[PASS] test_setup() (gas: 10630)
[PASS] test_splitFunds() (gas: 126130)
[PASS] test_split_funds_rounding() (gas: 130711)
[PASS] test_validateSkipProvisioning_reverts() (gas: 29539)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 235.05ms (202.29ms CPU time)

Ran 14 tests for test/unit/PufferModuleManager.t.sol:PufferModuleManagerTest
[PASS] test_beaconUpgrade() (gas: 1605162)
[PASS] test_callDelegateTo(bytes32,address,(bytes,uint256),bytes32) (runs: 256, : 1581812, ~: 1581710)
[PASS] test_callUndelegate(bytes32) (runs: 256, : 1560401, ~: 1560401)
[PASS] test_callVerifyWithdrawalCredentials(bytes32) (runs: 256, : 1555808, ~: 1555808)
[PASS] test_callWithdrawNonBeaconChainETHBalanceWei(bytes32) (runs: 256, : 1551794, ~: 1551794)
[PASS] test_collect_rewards(bytes32) (runs: 256, : 2485098, ~: 2485098)
[PASS] test_completeQueuedWithdrawals(bytes32) (runs: 256, : 1561482, ~: 1561482)
[PASS] test_createPufferModule(bytes32) (runs: 256, : 1530997, ~: 1530997)
[PASS] test_donation(bytes32) (runs: 256, : 1537154, ~: 1537154)
[PASS] test_module_has_eigenPod(bytes32) (runs: 256, : 1533101, ~: 1533101)
[PASS] test_postRewardsRoot(bytes32,uint256) (runs: 256, : 1629329, ~: 1629329)
[PASS] test_rewards_claiming_from_eigenlayer(bytes32) (runs: 256, : 1558136, ~: 1558136)
[PASS] test_updateAVSRegistrationSignatureProof() (gas: 262553)
[PASS] test_verifyAndProcessWithdrawals(bytes32) (runs: 256, : 1567564, ~: 1567564)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 1.68s (4.47s CPU time)

Ran 49 tests for test/unit/PufferProtocol.t.sol:PufferProtocolTest
[PASS] test_batch_claim() (gas: 1040416)
[PASS] test_batch_vs_multiple_single_withdrawals() (gas: 130730396)
[PASS] test_burst_threshold() (gas: 1585694)
[PASS] test_changeMinimumVTAmount() (gas: 39349)
[PASS] test_claim_bond_for_single_withdrawal() (gas: 709477)
[PASS] test_create_existing_module_fails() (gas: 32999)
[PASS] test_create_puffer_module() (gas: 1523643)
[PASS] test_deposit_validator_tickets_approval() (gas: 256360)
[PASS] test_deposit_validator_tickets_permit_for_bob() (gas: 270256)
[PASS] test_different_amounts_batch_claim() (gas: 2041551)
[PASS] test_double_deposit_validator_tickets_approval() (gas: 274579)
[PASS] test_double_deposit_validator_tickets_permit_for_bob() (gas: 310562)
[PASS] test_double_withdrawal_reverts() (gas: 753214)
[PASS] test_empty_queue() (gas: 22490)
[PASS] test_fuzz_register_many_validators(uint8) (runs: 256, : 21127578, ~: 12801198)
[PASS] test_get_payload() (gas: 156397)
[PASS] test_new_vtPenalty_works() (gas: 1103067)
[PASS] test_provision_node() (gas: 5844850)
[PASS] test_provision_reverts() (gas: 54605)
[PASS] test_register_both_approve() (gas: 575507)
[PASS] test_register_both_permit() (gas: 629372)
[PASS] test_register_invalid_bls_key() (gas: 57575)
[PASS] test_register_invalid_privKey_shares() (gas: 96895)
[PASS] test_register_invalid_pubkey_shares_length() (gas: 100023)
[PASS] test_register_multiple_validators_and_skipProvisioning(bytes32,bytes32) (runs: 256, : 1894978, ~: 1894978)
[PASS] test_register_no_sgx() (gas: 441984)
[PASS] test_register_pufETH_approve_buy_VT() (gas: 531263)
[PASS] test_register_pufETH_pay_vt_approve() (gas: 546978)
[PASS] test_register_pufETH_permit_pay_VT() (gas: 558323)
[PASS] test_register_skip_provision_withdraw_vt() (gas: 572272)
[PASS] test_register_to_invalid_module() (gas: 90983)
[PASS] test_register_validator_key() (gas: 474798)
[PASS] test_register_validator_key_with_permit_reverts_invalid_vt_amount() (gas: 338353)
[PASS] test_register_validator_with_huge_commitment() (gas: 68022)
[PASS] test_register_with_non_whole_amount() (gas: 471653)
[PASS] test_setVTPenalty() (gas: 41087)
[PASS] test_setup() (gas: 27577)
[PASS] test_single_withdrawal() (gas: 1089414)
[PASS] test_skip_provisioning() (gas: 1000688)
[PASS] test_slashing_case_1() (gas: 917660)
[PASS] test_slashing_case_2() (gas: 924456)
[PASS] test_slashing_case_3() (gas: 940673)
[PASS] test_slashing_case_4() (gas: 962693)
```

```
[PASS] test_slashing_case_5() (gas: 955586)
[PASS] test_upgrade() (gas: 4254153)
[PASS] test_validator_griefing_attack() (gas: 945468)
[PASS] test_validator_limit_per_module() (gas: 523986)
[PASS] test_vt_withdrawals_after_batch_claim() (gas: 1052320)
[PASS] test_withdraw_vt_before_provisioning() (gas: 485537)
Suite result: ok. 49 passed; 0 failed; 0 skipped; finished in 13.76s (17.93s CPU time)

Ran 7 test suites in 13.94s (15.99s CPU time): 99 tests passed, 0 failed, 0 skipped (99 total tests)
```

8.2.2 pufETH

```
> forge test -vvv --match-path './test/unit/*'

Ran 28 tests for test/unit/PufETH.t.sol:PufETHTest
[PASS] testFail_redeem((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 628814, ~: 630088)
[PASS] testFail_withdraw((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 632579, ~: 635685)
[PASS] test_RT_deposit_redeem((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2369, ~: 2369)
[PASS] test_RT_deposit_withdraw((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2391, ~: 2391)
[PASS] test_RT_mint_redeem((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2347, ~: 2347)
[PASS] test_RT_mint_withdraw((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2391, ~: 2391)
[PASS] test_RT_redeem_deposit((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2392, ~: 2392)
[PASS] test_RT_redeem_mint((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2346, ~: 2346)
[PASS] test_RT_withdraw_deposit((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2389, ~: 2389)
[PASS] test_RT_withdraw_mint((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2347, ~: 2347)
[PASS] test_asset((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 479760, ~: 483482)
[PASS] test_convertToAssets((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 490281, ~: 492144)
[PASS] test_convertToShares((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 489344, ~: 491872)
[PASS] test_deposit((address[4],uint256[4],uint256[4],int256),uint256,uint256) (runs: 256, : 532550, ~: 536049)
[PASS] test_erc4626_interface() (gas: 238648)
[PASS] test_maxDeposit((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 480452, ~: 483470)
[PASS] test_maxMint((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 481138, ~: 483405)
[PASS] test_maxRedeem((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 480313, ~: 483658)
[PASS] test_maxWithdraw((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 483174, ~: 486769)
[PASS] test_mint((address[4],uint256[4],uint256[4],int256),uint256,uint256) (runs: 256, : 538827, ~: 542216)
[PASS] test_previewDeposit((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 530667, ~: 532835)
[PASS] test_previewMint((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 537485, ~: 539146)
[PASS] test_previewRedeem((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2390, ~: 2390)
[PASS] test_previewWithdraw((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 2346, ~: 2346)
[PASS] test_redeem((address[4],uint256[4],uint256[4],int256),uint256,uint256) (runs: 256, : 2345, ~: 2345)
[PASS] test_roles_setup() (gas: 99794)
[PASS] test_totalAssets((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 482964, ~: 485613)
[PASS] test_withdraw((address[4],uint256[4],uint256[4],int256),uint256,uint256) (runs: 256, : 2369, ~: 2369)
Suite result: ok. 28 passed; 0 failed; 0 skipped; finished in 16.38s (16.92s CPU time)

Ran 13 tests for test/unit/Timelock.t.sol:TimelockTest
[PASS] test_cancel_reverts_if_caller_unauthorized(address) (runs: 256, : 11272, ~: 11272)
[PASS] test_cancel_transaction() (gas: 38801)
[PASS] test_change_pauser() (gas: 23527)
[PASS] test_execute_reverts_if_caller_unauthorized(address) (runs: 256, : 13166, ~: 13166)
[PASS] test_initial_access_manager_setup(address) (runs: 256, : 68521, ~: 68521)
[PASS] test_pause_depositor(address) (runs: 256, : 59282, ~: 59282)
[PASS] test_pause_should_revert_if_bad_caller(address) (runs: 256, : 15709, ~: 15709)
[PASS] test_queue_should_revert_if_operations_is_not_the_caller(address) (runs: 256, : 11273, ~: 11273)
[PASS] test_queueing_duplicate_transaction_different_operation_id() (gas: 79063)
[PASS] test_setDelay_reverts_if_caller_unauthorized(address) (runs: 256, : 9506, ~: 9506)
[PASS] test_setPauser_reverts_if_caller_unauthorized(address) (runs: 256, : 9545, ~: 9545)
[PASS] test_set_delay_queued() (gas: 43164)
[PASS] test_update_delay_from_community_without_timelock() (gas: 31482)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 16.38s (554.89ms CPU time)
```

```
Ran 26 tests for test/unit/PufferVaultV2Property.t.sol:PufferVaultV2Property
[PASS] testFail_redeem((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 662370, ~: 662880)
[PASS] testFail_withdraw((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 670690, ~: 670472)
[PASS] test_RT_deposit_redeem((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 569296, ~: 569472)
[PASS] test_RT_deposit_withdraw((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 569771, ~: 570055)
[PASS] test_RT_mint_redeem((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 576894, ~: 577503)
[PASS] test_RT_mint_withdraw((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 577179, ~: 577250)
[PASS] test_RT_redeem_deposit((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 570106, ~: 570382)
[PASS] test_RT_redeem_mint((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 571773, ~: 572116)
[PASS] test_RT_withdraw_deposit((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 576218, ~: 576391)
[PASS] test_RT_withdraw_mint((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 577938, ~: 578115)
[PASS] test_asset((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 485968, ~: 486415)
[PASS] test_convertToAssets((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 497290, ~: 497572)
[PASS] test_convertToShares((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 497423, ~: 497464)
[PASS] test_deposit((address[4],uint256[4],uint256[4],int256),uint256,uint256) (runs: 256, : 541984, ~: 542755)
[PASS] test_maxDeposit((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 486323, ~: 486392)
[PASS] test_maxMint((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 486171, ~: 486312)
[PASS] test_maxRedeem((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 495368, ~: 495448)
[PASS] test_maxWithdraw((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 495525, ~: 495516)
[PASS] test_mint((address[4],uint256[4],uint256[4],int256),uint256,uint256) (runs: 256, : 550064, ~: 550267)
[PASS] test_previewDeposit((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 540344, ~: 540812)
[PASS] test_previewMint((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 548201, ~: 548526)
[PASS] test_previewRedeem((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 558823, ~: 558969)
[PASS] test_previewWithdraw((address[4],uint256[4],uint256[4],int256),uint256) (runs: 256, : 564879, ~: 565034)
[PASS] test_redeem((address[4],uint256[4],uint256[4],int256),uint256,uint256) (runs: 256, : 560359, ~: 560641)
[PASS] test_totalAssets((address[4],uint256[4],uint256[4],int256)) (runs: 256, : 489937, ~: 489956)
[PASS] test_withdraw((address[4],uint256[4],uint256[4],int256),uint256,uint256) (runs: 256, : 566441, ~: 566238)
Suite result: ok. 26 passed; 0 failed; 0 skipped; finished in 16.38s (96.95s CPU time)

Ran 3 test suites in 16.43s (49.15s CPU time): 67 tests passed, 0 failed, 0 skipped (67 total tests)
```

Remarks about Puffer test suite

The **Puffer** team has diligently crafted test suites that encapsulate the fundamental workflows for interacting with the protocol. It is worth acknowledging that the test suite is currently not comprehensive enough to encompass certain edge cases and functionalities, as evidenced by some noticeable issues that should be easily caught by unit tests. These areas may seem straightforward in terms of the application's behaviour, yet it remains a best practice to include them for a robust test suite.

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 procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, 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.