Security Audit Report for Puffer Fast Path Contracts

Date: September 3, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Puffer Finance
Target	Puffer Fast Path Contracts

Version History

Version	Date	Description
1.0	September 3, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-09-04 09:33:51+0000

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

Chapter 1 Introduction

1.1 About Target Contracts

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

The focus of this audit is on Puffer Fast Path Contracts [1] of Puffer Finance. The feature of Puffer Fast Path Contracts is designed to help the SNO (Staking Node Operators) to cheaply and quickly withdraw their rewards on the L2 network. The contracts covered in this audit include:

```
mainnet-contracts/script/DeployFWR.s.sol
mainnet-contracts/src/PufferVaultV3.sol
mainnet-contracts/src/L1RewardManager.sol
l2-contracts/src/L2RewardManager.sol
mainnet-contracts/src/RestakingOperator.sol
mainnet-contracts/src/PufferModuleManager.sol
mainnet-contracts/src/PufferModule.sol
```

Listing 1.1: Audit Scope for this Report

It is important to note that the scope does not cover full files for the RestakingOperator, PufferModuleManager, and PufferModule contracts. We merely focus on code updates of these contracts to support the EigenLayer M4 upgrade.

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

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (Version 1), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Puffer Fast Path Contracts	Version 1	f058bb757636e754420a7d350ee561b087242ce2
	Version 2	1748779b57c2b8e14f7c860eae1b99cdbf8550b7

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,

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

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 [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

Table 1.1: Vulnerability Severity Classification

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

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

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

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.

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 found **eight** potential security issues. Besides, we have **four** recommendations and **one** note.

- Medium Risk: 3
- Low Risk: 5
- Recommendation: 4
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Possible overflow in function _handleMintAndBridge()	Software Security	Fixed
2	Medium	Potential stuck of xPufETH when reverting mint rewards	DeFi Security	Fixed
3	Medium	Lack of check on freeze status in function revertInterval()	DeFi Security	Fixed
4	Medium	Incorrect checks in function _setClaimingDelay()	DeFi Security	Fixed
5	Low	Possible incorrect destinationDomainId for function _revertInterval()	DeFi Security	Confirmed
6	Low	Lack of checks on allowedRewardMintFrequency	DeFi Security	Fixed
7	Low	Users receive less xPufETH after the mint rewards reversion	DeFi Security	Confirmed
8	Low	Lack of check on origin domain in function xReceive()	DeFi Security	Fixed
9	-	Use returned identifier from function createSelectFork()	Recommendation	Fixed
10	-	Handle possibly dust xPufETH tokens in contract L2RewardManager	Recommendation	Confirmed
11	-	Ensure epochRecord exists in function claimRewards()	Recommendation	Fixed
12	-	Possible incorrect delegate in function setL2RewardClaimer()	Recommendation	Fixed
13	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Possible overflow in function _handleMintAndBridge()

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The function _handleMintAndBridge() in the L2RewardManager contract contains a potential overflow issue when dealing with the exchange rate exceeding `uint(64).max`. Although the likelihood of this extreme case may be minimal, it is advised to address such edge cases.

```
function _handleMintAndBridge(uint256 amount, bytes memory data) internal {
    L1RewardManagerStorage.MintAndBridgeData memory params =
        abi.decode(data, (L1RewardManagerStorage.MintAndBridgeData));

    // Sanity check
    if (amount != ((params.rewardsAmount * params.ethToPufETHRate) / 1 ether)) {
        revert InvalidAmount();
    }

    RewardManagerStorage storage $ = _getRewardManagerStorage();

    bytes32 intervalId = getIntervalId(params.startEpoch, params.endEpoch);

    $.currentRewardsInterval = intervalId;

    $.epochRecords[intervalId] = EpochRecord({
        ethToPufETHRate: uint64(params.ethToPufETHRate),
        startEpoch: uint72(params.startEpoch),
        endEpoch: uint72(params.endEpoch),
        timeBridged: uint48(block.timestamp),
        rewardRoot: params.rewardsRoot,
        pufETHAmount: uint128(amount),
        ethAmount: uint128(params.rewardsAmount)
    });

    emit RewardRootAndRatePosted({
        rewardsAmount: params.rewardsAmount,
        ethToPufETHRate: params.ethToPufETHRate,
        startEpoch: params.startEpoch,
        intervalId: intervalId,
        endEpoch: params.endEpoch,
        rewardsRoot: params.rewardsRoot
    });
}
```

Listing 2.1: l2-contracts/src/L2RewardManager.sol

Impact Potential incorrect `ethToPufETHRate` recorded on L2.

Suggestion Use type `uint256` for `EpochRecord.ethToPufETHRate`.

2.2 DeFi Security

2.2.1 Potential stuck of `xPufETH` when reverting mint rewards

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The xReceive() function in the L1RewardManager contract invokes the function revertMintRewards() in PufferVaultV3 after receiving the xPufETH bridged from the L2RewardManager. The function revertMintRewards() will decrease the totalRewardMintAmount by the specified revert amount. However, if rewards have already been distributed through the depositRewards() function, the totalRewardMintAmount might be insufficient, leading to a revert due to an underflow issue.

Furthermore, according to the bridge's documentation [1], if the call on the receiver contract (also referred to as the 'target' contract) reverts, funds sent with the call will remain in the receiver contract. In this case, the xPufETH will get stuck in the L1RewardManager contract.

```
function xReceive(bytes32, uint256, address, address originSender, uint32, bytes memory callData)
    external
    override(IXReceiver)
    restricted
    returns (bytes memory)
{
    // The call must originate from the L2_REWARDS_MANAGER
    if (originSender != address(L2_REWARDS_MANAGER)) {
        revert Unauthorized();
    }

    // We decode the data to get the amount of shares(pufETH) and the ETH amount.
    L2RewardManagerStorage.EpochRecord memory epochRecord =
        abi.decode(callData, (L2RewardManagerStorage.EpochRecord));

    XPUFETH.approve(address(LOCKBOX), epochRecord.pufETHAmount);
    // get the pufETH
    LOCKBOX.withdraw(epochRecord.pufETHAmount);

    // The PufferVault will burn the pufETH from this contract and subtract the ETH amount from the ethRewardsAmount
    PUFFER_VAULT.revertMintRewards({ pufETHAmount: epochRecord.pufETHAmount, ethAmount: epochRecord.ethAmount });

    emit RevertedRewards({
        rewardsAmount: epochRecord.ethAmount,
        startEpoch: epochRecord.startEpoch,
        endEpoch: epochRecord.endEpoch,
        rewardsRoot: epochRecord.rewardRoot
    });

    return "";
}
```

Listing 2.2: mainnet-contracts/src/L1RewardManager.sol

```
function depositRewards() external payable restricted {
    VaultStorage storage $ = _getPufferVaultStorage();
    uint256 previousRewardsAmount = $.totalRewardMintAmount;
```

[1] https://docs.connext.network/developers/guides/handling-failures

```
    uint256 newTotalRewardsAmount = previousRewardsAmount - msg.value;
    $.totalRewardMintAmount = newTotalRewardsAmount;

    emit UpdatedTotalRewardsAmount(previousRewardsAmount, newTotalRewardsAmount, msg.value);
}

/**
 * @notice Reverts the ‘mintRewards‘ action.
 * @dev Restricted to L1RewardManager
 */
function revertMintRewards(uint256 pufETHAmount, uint256 ethAmount) external restricted {
    VaultStorage storage $ = _getPufferVaultStorage();

    uint256 previousRewardsAmount = $.totalRewardMintAmount;
    uint256 newTotalRewardsAmount = previousRewardsAmount - ethAmount;
    $.totalRewardMintAmount = newTotalRewardsAmount;

    emit UpdatedTotalRewardsAmount(previousRewardsAmount, newTotalRewardsAmount, 0);

    // msg.sender is the L1RewardManager contract
    _burn(msg.sender, pufETHAmount);
}
```

Listing 2.3: mainnet-contracts/src/PufferVaultV3.sol

Impact Bridged xPufETH tokens could get stuck in the L1RewardManager contract, resulting in an inconsistent state and causing some users to lose their rewards.

Suggestion Revise the function depositRewards() to maintain a separate variable that tracks already distributed rewards.

2.2.2 Lack of check on freeze status in function revertInterval()

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The function revertInterval() in the L2RewardManager contract does not check the interval's freeze status. This is inconsistent with the code annotation on line 165, which suggests that only a frozen interval can be reverted. Moreover, while the function _freezeClaimingForInterval() forbids reverting an unlocked interval (lines 328-331), revertInterval() omits such sanity checks. Consequently, an unlocked interval can also be reverted, leading to inconsistencies between L1 and L2 accounting.

```
164   /**
165   * @notice Reverts the already frozen interval. It bridges the xPufETH back to the L1
166   * @dev On the L1, we unwrap xPufETH to pufETH and burn the pufETH to undo the minting
167   * We use msg.value to pay for the relayer fee on the destination chain.
168   */
169  function revertInterval(address bridge, uint256 startEpoch, uint256 endEpoch) external payable
          restricted {
170      _revertInterval(bridge, startEpoch, endEpoch);
}
```

Listing 2.4: l2-contracts/src/L2RewardManager.sol

```
function _freezeClaimingForInterval(uint256 startEpoch, uint256 endEpoch) internal {
    RewardManagerStorage storage $ = _getRewardManagerStorage();

    bytes32 intervalId = getIntervalId(startEpoch, endEpoch);

    // Revert if the claiming is not locked
    if (!_isClaimingLocked(intervalId)) {
        revert UnableToFreezeInterval();
    }

    // revert for non-existing interval
    if ($.epochRecords[intervalId].rewardRoot == bytes32(0)) {
        revert UnableToFreezeInterval();
    }

    // To freeze the claiming, we set the timeBridged to 0
    $.epochRecords[intervalId].timeBridged = 0;

    emit ClaimingIntervalFrozen({ startEpoch: startEpoch, endEpoch: endEpoch });
}
```

Listing 2.5: l2-contracts/src/L2RewardManager.sol

```
function _revertInterval(address bridge, uint256 startEpoch, uint256 endEpoch) internal {
    RewardManagerStorage storage $ = _getRewardManagerStorage();

    BridgeData memory bridgeData = $.bridges[bridge];

    if (bridgeData.destinationDomainId == 0) {
        revert BridgeNotAllowlisted();
    }

    bytes32 intervalId = getIntervalId(startEpoch, endEpoch);

    EpochRecord memory epochRecord = $.epochRecords[intervalId];

    XPUFETH.approve(bridge, epochRecord.pufETHAmount);

    IBridgeInterface(bridge).xcall{ value: msg.value }({
        destination: bridgeData.destinationDomainId, // Domain ID of the destination chain
        to: L1_REWARD_MANAGER, // Address of the target contract
        asset: address(XPUFETH), // Address of the token contract
        delegate: msg.sender, // Address that can revert or forceLocal on destination
        amount: epochRecord.pufETHAmount, // Amount of tokens to transfer
        slippage: 0, // Max slippage the user will accept in BPS (e.g. 300 = 3%)
        callData: abi.encode(epochRecord) // Encoded data to send
     });

    delete $.epochRecords[intervalId];

    emit ClaimingIntervalReverted({
        startEpoch: startEpoch,
        endEpoch: endEpoch,
        intervalId: intervalId,
        pufETHAmount: epochRecord.pufETHAmount,
        rewardsRoot: epochRecord.rewardRoot
    });
}
```

Listing 2.6: l2-contracts/src/L2RewardManager.sol

Impact Reverts on unfrozen intervals could potentially disrupt the accounting and reward functionality.

Suggestion Add a check to ensure the interval is frozen (i.e., `timeBridged == 0 && rewardRoot != bytes32(0)`) in the function `revertInterval()`.

2.2.3 Incorrect checks in function `_setClaimingDelay()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_setClaimingDelay()` will revert if the current claiming interval is not locked and the new delayed timestamp (i.e., `timeBridged + newDelay`) exceeds the current timestamp. However, when the current claiming interval is locked, the `newDelay` can be set to any value more than 6 hours, which may make older unlocked claiming intervals being locked again.

```
function _setClaimingDelay(uint256 newDelay) internal {
    if (newDelay < 6 hours) {
        revert InvalidDelayPeriod();
    }
    RewardManagerStorage storage $ = _getRewardManagerStorage();

    // Revert only if the claiming is not locked and the new delayed timestamp (timeBridged+
        newDelay) exceedes the current timestamp
    if (
        !_isClaimingLocked($.currentRewardsInterval)
            && ($.epochRecords[$.currentRewardsInterval].timeBridged + newDelay > block.
                timestamp)
    ) {
        revert RelockingIntervalIsNotAllowed();
    }

    emit ClaimingDelayChanged({ oldDelay: $.claimingDelay, newDelay: newDelay });
    $.claimingDelay = newDelay;
}
```

Listing 2.7: l2-contracts/src/L2RewardManager.sol

Impact Unlocked claiming intervals can be changed to locked.

Suggestion Track every interval's unlocked time using a separate data struct (e.g., mapping) and invoking function `_setClaimingDelay()` will only affect current and future claiming intervals.

2.2.4 Possible incorrect `destinationDomainId` for function `_revertInterval()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the current implementation, the `_revertInterval` function in the `L2RewardManager` contract initiates a cross-chain call using the `bridgeData.destinationDomainId` parameter. However, the function `updateBridgeData()` allows the `destinationDomainId` for a bridge to be updated, which could result in the cross-chain revert call being accidentally bridged to an incorrect target chain instead of the original one.

```
function updateBridgeData(address bridge, BridgeData memory bridgeData) external restricted {
    RewardManagerStorage storage $ = _getRewardManagerStorage();

    if (bridge == address(0)) {
        revert InvalidAddress();
    }

    $.bridges[bridge].destinationDomainId = bridgeData.destinationDomainId;
    emit BridgeDataUpdated(bridge, bridgeData);
}
```

Listing 2.8: l2-contracts/src/L2RewardManager.sol

```
function _revertInterval(address bridge, uint256 startEpoch, uint256 endEpoch) internal {
    RewardManagerStorage storage $ = _getRewardManagerStorage();

    BridgeData memory bridgeData = $.bridges[bridge];

    if (bridgeData.destinationDomainId == 0) {
        revert BridgeNotAllowlisted();
    }

    bytes32 intervalId = getIntervalId(startEpoch, endEpoch);

    EpochRecord memory epochRecord = $.epochRecords[intervalId];

    XPUFETH.approve(bridge, epochRecord.pufETHAmount);

    IBridgeInterface(bridge).xcall{ value: msg.value }({
        destination: bridgeData.destinationDomainId, // Domain ID of the destination chain
        to: L1_REWARD_MANAGER, // Address of the target contract
        asset: address(XPUFETH), // Address of the token contract
        delegate: msg.sender, // Address that can revert or forceLocal on destination
        amount: epochRecord.pufETHAmount, // Amount of tokens to transfer
        slippage: 0, // Max slippage the user will accept in BPS (e.g. 300 = 3%)
        callData: abi.encode(epochRecord) // Encoded data to send
     });

    delete $.epochRecords[intervalId];

    emit ClaimingIntervalReverted({
        startEpoch: startEpoch,
        endEpoch: endEpoch,
        intervalId: intervalId,
        pufETHAmount: epochRecord.pufETHAmount,
        rewardsRoot: epochRecord.rewardRoot
    });
}
```

Listing 2.9: l2-contracts/src/L2RewardManager.sol

Impact The cross-chain message could be bridged to the incorrect chain, leading to unexpected results.

Suggestion Record the original `destinationDomainId` in the `$.epochRecords` for each interval.

Feedback from the project The domainId doesn't change (Connext is using a constant for it). We added this `updateBridgeData()` function to be able to swap out `Connext` (in case some other bridge decides to use the same interface), although this is an unlikely scenario.

2.2.5 Lack of checks on `allowedRewardMintFrequency`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Currently, the function `mintAndBridgeRewards()` doesn't check whether `allowedRewardMintFrequency` is zero, which is not initialized in the function `initialize()`. If function `mintAndBridgeRewards()` is invoked before `allowedRewardMintFrequency` is initialized, the mint frequency check will be bypassed and lead to unexpected results.

Additionally, the function `setAllowedRewardMintFrequency()` doesn't ensure that the new frequency is bigger than a minimal value, which may bring possible risks.

```
function mintAndBridgeRewards(MintAndBridgeParams calldata params) external payable restricted
     {
    RewardManagerStorage storage $ = _getRewardManagerStorage();

    if (params.rewardsAmount > $.allowedRewardMintAmount) {
        revert InvalidMintAmount();
    }

    if (($.lastRewardMintTimestamp + $.allowedRewardMintFrequency) > block.timestamp) {
        revert NotAllowedMintFrequency();
    }
```

Listing 2.10: mainnet-contracts/src/L1RewardManager.sol

```
function setAllowedRewardMintFrequency(uint104 newFrequency) external restricted {
    RewardManagerStorage storage $ = _getRewardManagerStorage();

    emit AllowedRewardMintFrequencyUpdated($.allowedRewardMintFrequency, newFrequency);

    $.allowedRewardMintFrequency = newFrequency;
}
```

Listing 2.11: src/rewards/MultiRewardDistributor.sol

Impact The frequency check in function `mintAndBridgeRewards()` can be ineffective.

Suggestion Add checks on `allowedRewardMintFrequency` in functions `mintAndBridgeRewards()` and `setAllowedRewardMintFrequency()` accordingly.

2.2.6 Users receive less `xPufETH` after the mint rewards reversion

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The `pufETH` token is an interest-bearing token that is expected to increase in value as `PufferVault` receives validator rewards. The function `mintRewards()` in the `PufferVaultV3` contract calculates the `pufETHAmount` using the spot price returned by `convertToShares(1 ether)`. The function `revertMintRewards()` burns a corresponding `pufETHAmount` of `pufETH` tokens, based on the exchange rate at the time of minting. Due to the potential increase in `pufETH`'s price between the minting and reverting periods, users may lose a portion of their rewards when the rewards are re-minted and bridged.

```
function mintRewards(uint256 rewardsAmount)
    external
    restricted
    returns (uint256 ethToPufETHRate, uint256 pufETHAmount)
{
    ethToPufETHRate = convertToShares(1 ether);
    // calculate the shares using this formula since calling convertToShares again is costly
    pufETHAmount = ethToPufETHRate.mulDiv(rewardsAmount, 1 ether, Math.Rounding.Floor);

    VaultStorage storage $ = _getPufferVaultStorage();

    uint256 previousRewardsAmount = $.totalRewardMintAmount;
    uint256 newTotalRewardsAmount = previousRewardsAmount + rewardsAmount;
    $.totalRewardMintAmount = newTotalRewardsAmount;

    emit UpdatedTotalRewardsAmount(previousRewardsAmount, newTotalRewardsAmount, 0);

    // msg.sender is the L1RewardManager contract
    _mint(msg.sender, pufETHAmount);

    return (ethToPufETHRate, pufETHAmount);
}
```

Listing 2.12: mainnet-contracts/src/PufferVaultV3.sol

```solidity
function xReceive(bytes32, uint256, address, address originSender, uint32, bytes memory callData)
    external
    override(IXReceiver)
    restricted
    returns (bytes memory)
{
    // The call must originate from the L2_REWARDS_MANAGER
    if (originSender != address(L2_REWARDS_MANAGER)) {
        revert Unauthorized();
    }

    // We decode the data to get the amount of shares(pufETH) and the ETH amount.
    L2RewardManagerStorage.EpochRecord memory epochRecord =
        abi.decode(callData, (L2RewardManagerStorage.EpochRecord));

    XPUFETH.approve(address(LOCKBOX), epochRecord.pufETHAmount);
    // get the pufETH
    LOCKBOX.withdraw(epochRecord.pufETHAmount);

    // The PufferVault will burn the pufETH from this contract and subtract the ETH amount from the ethRewardsAmount
    PUFFER_VAULT.revertMintRewards({ pufETHAmount: epochRecord.pufETHAmount, ethAmount: epochRecord.ethAmount });

    emit RevertedRewards({
        rewardsAmount: epochRecord.ethAmount,
        startEpoch: epochRecord.startEpoch,
        endEpoch: epochRecord.endEpoch,
        rewardsRoot: epochRecord.rewardRoot
    });

    return "";
}
```

Listing 2.13: mainnet-contracts/src/L1RewardManager.sol

```solidity
function revertMintRewards(uint256 pufETHAmount, uint256 ethAmount) external restricted {
    VaultStorage storage $ = _getPufferVaultStorage();

    uint256 previousRewardsAmount = $.totalRewardMintAmount;
    uint256 newTotalRewardsAmount = previousRewardsAmount - ethAmount;
    $.totalRewardMintAmount = newTotalRewardsAmount;

    emit UpdatedTotalRewardsAmount(previousRewardsAmount, newTotalRewardsAmount, 0);

    // msg.sender is the L1RewardManager contract
    _burn(msg.sender, pufETHAmount);
}
```

Listing 2.14: mainnet-contracts/src/PufferVaultV3.sol

Impact Staking node operators may receive less `xPufETH` when rewards are redistributed.

Suggestion Revise the logic to ensure the correct distribution of rewards for each interval.

Feedback from the project We acknowledge this issue and we will find a way to reimburse the node operators if this scenario happens. Coding the logic in the smart contract, would add complexity which we do not want.

2.2.7 Lack of check on origin domain in function `xReceive()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In both `L1RewardManager` and `L2RewardManager` contracts, the function `xReceive()` should verify the validity of the origin data, as Connext's documentation [2] suggests. However, the current implementation lacks validation on the `_origin`, i.e., the origin domain ID.

```
function xReceive(bytes32, uint256, address, address originSender, uint32, bytes memory
     callData)
    external
    override(IXReceiver)
    restricted
    returns (bytes memory)
{
    // The call must originate from the L2_REWARDS_MANAGER
    if (originSender != address(L2_REWARDS_MANAGER)) {
        revert Unauthorized();
    }

    // We decode the data to get the amount of shares(pufETH) and the ETH amount.
    L2RewardManagerStorage.EpochRecord memory epochRecord =
        abi.decode(callData, (L2RewardManagerStorage.EpochRecord));

    XPUFETH.approve(address(LOCKBOX), epochRecord.pufETHAmount);
    // get the pufETH
    LOCKBOX.withdraw(epochRecord.pufETHAmount);

    // The PufferVault will burn the pufETH from this contract and subtract the ETH amount from
         the ethRewardsAmount
    PUFFER_VAULT.revertMintRewards({ pufETHAmount: epochRecord.pufETHAmount, ethAmount:
         epochRecord.ethAmount });

    emit RevertedRewards({
        rewardsAmount: epochRecord.ethAmount,
        startEpoch: epochRecord.startEpoch,
        endEpoch: epochRecord.endEpoch,
```

[2] https://docs.connext.network/developers/guides/authentication

```
        rewardsRoot: epochRecord.rewardRoot
    });

    return "";
}
```

Listing 2.15: mainnet-contracts/src/L1RewardManager.sol

```
function xReceive(bytes32, uint256 amount, address, address originSender, uint32, bytes memory
      callData)
    external
    override(IXReceiver)
    restricted
    returns (bytes memory)
{
    if (originSender != address(L1_REWARD_MANAGER)) {
        revert Unauthorized();
    }

    IL1RewardManager.BridgingParams memory bridgingParams = abi.decode(callData, (
         IL1RewardManager.BridgingParams));

    if (bridgingParams.bridgingType == IL1RewardManager.BridgingType.MintAndBridge) {
        _handleMintAndBridge(amount, bridgingParams.data);
    } else if (bridgingParams.bridgingType == IL1RewardManager.BridgingType.SetClaimer) {
        _handleSetClaimer(bridgingParams.data);
    }
    // Return empty data
    return "";
}
```

Listing 2.16: mainnet-contracts/src/L1RewardManager.sol

Impact Executing calls from unexpected origin domains may disrupt the functionality.

Suggestion Add a whitelist to hold supported domains and check `_origin` against the whitelist.

2.3 Additional Recommendation

2.3.1 Use returned identifier from function `createSelectFork()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It's recommended to use the returned identifier from the function `createSelectFork()` as the parameter for the function `selectFork()`, rather than use a hardcoded zero value in contract `DeployFWR`.

```
function run() public {
    GenerateAccessManagerCalldata3 generator = new GenerateAccessManagerCalldata3();

```

```
        vm.createSelectFork(vm.rpcUrl("mainnet"));

        vm.startBroadcast();

        address noImpl = address(new NoImplementation());

        // Deploy empty proxy
        l1RewardManagerProxy = address(new ERC1967Proxy(noImpl, ""));

        vm.label(address(l1RewardManagerProxy), "l1RewardManagerProxy");

        // Generate L1 calldata
        bytes memory l1AccessManagerCalldata = generator.generateL1Calldata({
            l1RewardManagerProxy: l1RewardManagerProxy,
            l1Bridge: _getEverclear(),
            pufferVaultProxy: _getPufferVault(),
            pufferModuleManagerProxy: _getPufferModuleManager()
        });

        console.log("L1 Access Manager Calldata");
        console.logBytes(l1AccessManagerCalldata);

        vm.stopBroadcast();

        // Deploy contracts on L2
        vm.createSelectFork(vm.rpcUrl("base"));
        vm.startBroadcast();

        L2RewardManager newImplementation = new L2RewardManager(_getXPufETH(), address(l1RewardManagerProxy));

        console.log("L2RewardManager Implementation", address(newImplementation));

        l2RewardManagerProxy = address(
            new ERC1967Proxy(
                address(newImplementation), abi.encodeCall(L2RewardManager.initialize, (_getAccessManager()))
            )
        );
        vm.makePersistent(l2RewardManagerProxy);

        console.log("L2RewardManager Proxy", address(l2RewardManagerProxy));
        vm.label(address(l2RewardManagerProxy), "L2RewardManagerProxy");
        vm.label(address(newImplementation), "L2RewardManagerImplementation");

        bytes memory l2AccessManagerCalldata =
            generator.generateL2Calldata({ l2RewardManagerProxy: l2RewardManagerProxy, l2Bridge: _getEverclear() });

        console.log("L2 Access Manager Calldata");
        console.logBytes(l2AccessManagerCalldata);

        // Upgrade contract on L1
        vm.stopBroadcast();

        // Switch back to Fork 0
        vm.selectFork(0);
        vm.startBroadcast();
```

Listing 2.17: mainnet-contracts/script/DeployFWR.s.sol

Suggestion Use the returned identifier from the function `createSelectFork()` for function `selectFork()`.

2.3.2 Handle possibly dust `xPufETH` tokens in contract `L2RewardManager`

Status Confirmed

Introduced by `Version 1`

Description In the `L1RewardManager` contract, the function `mintAndBridge()` calculates the `pufETHAmount` on Line 77 based on the total rewards for all validators (i.e., `ethToPufETHRate-.mulDiv(rewardsAmount, 1 ether, Math.Rounding.Floor)`). However, when users claim rewards on L2, each user's reward amount is calculated based on their individual share in the `claimRewards` function. The cumulative precision loss for each user can result in residual dust `xPufETH` tokens in the `L2RewardManager` contract.

```
70    function mintRewards(uint256 rewardsAmount)
71        external
72        restricted
73        returns (uint256 ethToPufETHRate, uint256 pufETHAmount)
74    {
75        ethToPufETHRate = convertToShares(1 ether);
76        // calculate the shares using this formula since calling convertToShares again is costly
77        pufETHAmount = ethToPufETHRate.mulDiv(rewardsAmount, 1 ether, Math.Rounding.Floor);
78
79        VaultStorage storage $ = _getPufferVaultStorage();
80
81        uint256 previousRewardsAmount = $.totalRewardMintAmount;
82        uint256 newTotalRewardsAmount = previousRewardsAmount + rewardsAmount;
83        $.totalRewardMintAmount = newTotalRewardsAmount;
84
85        emit UpdatedTotalRewardsAmount(previousRewardsAmount, newTotalRewardsAmount, 0);
86
87        // msg.sender is the L1RewardManager contract
88        _mint(msg.sender, pufETHAmount);
89
90        return (ethToPufETHRate, pufETHAmount);
91    }
```

Listing 2.18: mainnet-contracts/src/PufferVaultV3.sol

```
function claimRewards(ClaimOrder[] calldata claimOrders) external restricted {
    for (uint256 i = 0; i < claimOrders.length; i++) {
        if (isClaimed(claimOrders[i].intervalId, claimOrders[i].account)) {
            revert AlreadyClaimed(claimOrders[i].intervalId, claimOrders[i].account);
            }

            RewardManagerStorage storage $ = _getRewardManagerStorage();

            // L1 contracts MUST set the claimer
            address recipient = $.rewardsClaimers[claimOrders[i].account];
            if (claimOrders[i].isL1Contract && recipient == address(0)) {
                revert ClaimerNotSet(claimOrders[i].account);
            }

            EpochRecord storage epochRecord = $.epochRecords[claimOrders[i].intervalId];

            if (_isClaimingLocked(claimOrders[i].intervalId)) {
                revert ClaimingLocked({
                    intervalId: claimOrders[i].intervalId,
                    account: claimOrders[i].account,
                    lockedUntil: epochRecord.timeBridged + $.claimingDelay
                });
            }

            // Alice may run many Puffer validators in the same interval ‘totalETHEarned = sum(aliceValidators)‘
            // The leaf is: keccak256(abi.encode(AliceAddress, isL1Contract, totalETHEarned))
            bytes32 leaf = keccak256(
                bytes.concat(
                    keccak256(abi.encode(claimOrders[i].account, claimOrders[i].isL1Contract, claimOrders[i].amount))
                )
            );
            if (!MerkleProof.verifyCalldata(claimOrders[i].merkleProof, epochRecord.rewardRoot, leaf)) {
                revert InvalidProof();
            }

            // Mark it claimed and transfer the tokens
            $.claimedRewards[claimOrders[i].intervalId][claimOrders[i].account] = true;

            uint256 amountToTransfer = (claimOrders[i].amount * epochRecord.ethToPufETHRate) / 1 ether;

            recipient = recipient == address(0) ? claimOrders[i].account : recipient;

            // if the custom claimer is set, then transfer the tokens to the set claimer
            XPUFETH.transfer(recipient, amountToTransfer);

            emit Claimed({
                recipient: recipient,
                account: claimOrders[i].account,
                intervalId: claimOrders[i].intervalId,
                amount: amountToTransfer
            });
        }
    }
```

Listing 2.19: l2-contracts/src/L2RewardManager.sol

Suggestion Add a function to handle the dust `xPufETH` left in the `L2RewardManager` contract.

Feedback from the project Acknowledged. The contract is upgradeable, if the dust gets big enough, we will handle it in the future.

2.3.3 Ensure `epochRecord` exists in function `claimRewards()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `claimRewards()` doesn't check if `epochRecord` exists before invoking function `_isClaimingLocked()`, which may trigger a `ClaimingLocked` revert with wrong information.

```
function claimRewards(ClaimOrder[] calldata claimOrders) external restricted {
    for (uint256 i = 0; i < claimOrders.length; i++) {
        if (isClaimed(claimOrders[i].intervalId, claimOrders[i].account)) {
            revert AlreadyClaimed(claimOrders[i].intervalId, claimOrders[i].account);
        }

        RewardManagerStorage storage $ = _getRewardManagerStorage();

        // L1 contracts MUST set the claimer
        address recipient = $.rewardsClaimers[claimOrders[i].account];
        if (claimOrders[i].isL1Contract && recipient == address(0)) {
            revert ClaimerNotSet(claimOrders[i].account);
        }

        EpochRecord storage epochRecord = $.epochRecords[claimOrders[i].intervalId];

        if (_isClaimingLocked(claimOrders[i].intervalId)) {
            revert ClaimingLocked({
                intervalId: claimOrders[i].intervalId,
                account: claimOrders[i].account,
                lockedUntil: epochRecord.timeBridged + $.claimingDelay
            });
        }
```

Listing 2.20: l2-contracts/src/L2RewardManager.sol

Suggestion Check if the `epochRecord` exists and revert with a specific error.

2.3.4 Possible incorrect delegate in function `setL2RewardClaimer()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `delegate` parameter of the bridge call is set as the `msg.sender` in function `setL2RewardClaimer()`. However, the user (i.e., `msg.sender`) on L1 may not control the same address on L2, which may bring risks since the address of the delegate can update slippage and other bridge parameters on the destination chain.

```
function setL2RewardClaimer(address bridge, address claimer) external payable {
    RewardManagerStorage storage $ = _getRewardManagerStorage();

    BridgeData memory bridgeData = $.bridges[bridge];

    if (bridgeData.destinationDomainId == 0) {
        revert BridgeNotAllowlisted();
    }

    // msg.value is used to pay for the relayer fee on the destination chain
    IBridgeInterface(bridge).xcall{ value: msg.value }({
        destination: bridgeData.destinationDomainId, // Domain ID of the destination chain
        to: L2_REWARDS_MANAGER, // Address of the target contract on the destination chain
        delegate: msg.sender, // Address that can revert or forceLocal on destination
        asset: address(0), // Address of the token contract
        amount: 0, // We don't transfer any tokens
        slippage: 0, // No slippage
        callData: abi.encode(
            BridgingParams({
                bridgingType: BridgingType.SetClaimer,
                data: abi.encode(SetClaimerParams({ account: msg.sender, claimer: claimer }))
            })
        ) // Encoded data to bridge to the target contract
     });
```

Listing 2.21: mainnet-contracts/src/L1RewardManager.sol

Suggestion Let the user pass the delegate as a parameter of the function `setL2RewardClaimer()`, or use an address controlled by the puffer protocol.

2.4 Note

2.4.1 Potential centralization risks

Introduced by `Version 1`

Description In `puffer-contracts`, there are some privileged functions to update critical configurations, such as the bridge and mint configurations. These functions have restricted modifiers and are claimed to be controlled by a multi-signature wallet. If most of the private keys in this wallet are controlled by a single entity, or if the private keys are leaked. The protocol can be potentially incapacitated.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS