Security Review Report
NM-0245 POCKET NETWORK

NETHERMIND
SECURITY

(Jun 12, 2024)

Contents

1 Executive Summary **2**

2 Audited Files **3**

3 Summary of Issues **3**

4 System Overview **4**
4.1 xPOKT token . . . 4
4.2 WPOKT LockBox . . . 5
4.3 Bridging Mechanism . . . 5
4.3.1 Bridging `xPOKT` via `WormholeBridgeAdapter` . . . 5
4.3.2 Bridging `xPOKT` via `wPOKTRouter` . . . 5

5 Risk Rating Methodology **6**

6 Issues **7**
6.1 [Medium] Refund address is not specified . . . 7
6.2 [Medium] The `ownerUnpause()` function does not unpause the contract . . . 7
6.3 [Info] Allowing excessive funds in `_bridgeTo(...)` function is unnecessary . . . 8
6.4 [Info] Amount of gas can differ on various chains . . . 8

7 Documentation Evaluation **9**

8 Test Suite Evaluation **10**
8.1 Compilation Output . . . 10
8.2 Tests Output . . . 10

9 About Nethermind **12**

1 Executive Summary

This document outlines the security review conducted by Nethermind Security for the Pocket Network contracts. The review focuses on the new `xPOKT` token and its bridging mechanism. The `xPOKT` token is a cross-chain token designed to become the native Pocket Network token and will be deployed on different EVM chains. It inherits from the `xERC20` token contract and implements a mint limiting and pausing functionalities.

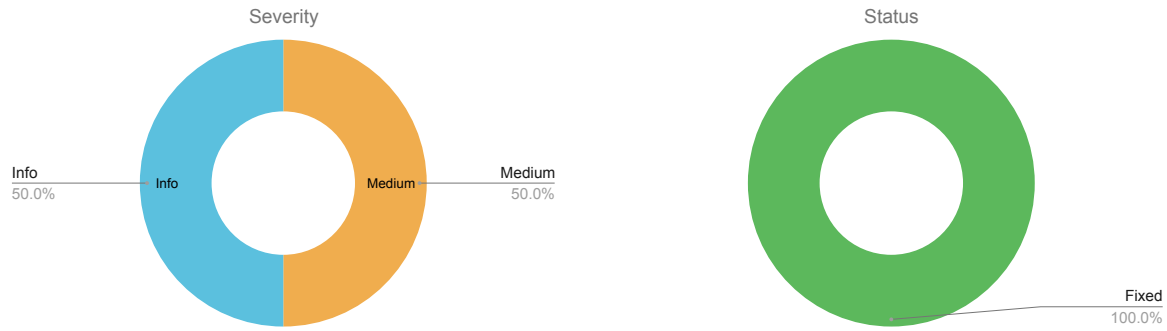
Currently, users hold `wPOKT` token on Ethereum network. To seamlessly transition to `xPOKT`, they should interact with the `XERC20Lockbox` contract, where they lock their `wPOKT` tokens and mint an equivalent amount of `xPOKT` tokens. Subsequently, `xPOKT` holders can bridge their assets to any supported EVM chain by initiating a bridge operation through the `WormholeBridgeAdapter` contract.

Moreover, holders of `wPOKT` tokens can leverage the `wPOKTRouter` to bridge their assets to another chain directly. This process involves implicitly locking the `wPOKT` tokens in the lockbox and invoking the `bridge(...)` function on the `WormholeBridgeAdapter` contract, thereby reducing the number of transactions required to complete the bridging operation.

The audited code comprises 738 lines of code in Solidity. The **Pocket Network** team has provided documentation that explains the purpose and functionality behind audited contracts.

The audit was performed using: (a) manual analysis of the codebase, (b) simulation of the smart contracts. **Along this document, we report** 4 points of attention, where two are classified as `Medium` and two are classified as `Informational`. 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** (0), **Medium** (2), **Low** (0), **Undetermined** (0), **Informational** (2), **Best Practices** (0). **(b) Distribution of status: Fixed** (4), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jun 12, 2024
Final Report	Jun 12, 2024
Methods	Manual Review, Automated analysis
Repository	pocket-contracts
Commit Hash	932d3f7da9837849bcffba31b95164d8f8f218d5
Final Commit Hash	be8d7ef84b551d544b9b5e14c1a5cbae989152ff
Documentation	HackMD
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	governance/WormholeTrustedSender.sol	83	39	47.0%	23	145
2	xPOKT/xPOKT.sol	144	89	61.8%	39	272
3	xPOKT/WPOKTRouter.sol	58	34	58.6%	22	114
4	xPOKT/WormholeBridgeAdapter.sol	125	94	75.2%	37	256
5	xPOKT/ConfigurablePauseGuardian.sol	48	35	72.9%	19	102
6	xPOKT/ConfigurablePause.sol	26	28	107.7%	12	66
7	xPOKT/MintLimits.sol	122	64	52.5%	33	219
8	xPOKT/xERC20BridgeAdapter.sol	51	38	74.5%	15	104
9	xPOKT/XERC20Lockbox.sol	36	22	61.1%	16	74
10	xPOKT/xERC20.sol	45	46	102.2%	19	110
	Total	**738**	**489**	**66.3%**	**235**	**1462**

3 Summary of Issues

	Finding	Severity	Update
1	Refund address is not specified	Medium	Fixed
2	The `ownerUnpause()` function does not unpause the contract	Medium	Fixed
3	Allowing excessive funds in `_bridgeTo(...)` function is unnecessary	Info	Fixed
4	Amount of gas can differ on various chains	Info	Fixed

4 System Overview

Pocket Network introduced the `xPOKT` token, a native cross-chain token that will be deployed across different EVM chains. Currently, liquidity on the Ethereum mainnet resides on the `wPOKT` token. Holders of `wPOKT` can transition to `xPOKT` via the `XERC20Lockbox` contract and then utilize the `WormholeBridgeAdapter` contract to bridge `xPOKT` to other EVM chains. Alternatively, `wPOKT` holders can directly leverage the `WPOKTRouter` for this purpose. Figure 1 shows a general overview of user flows within the system.

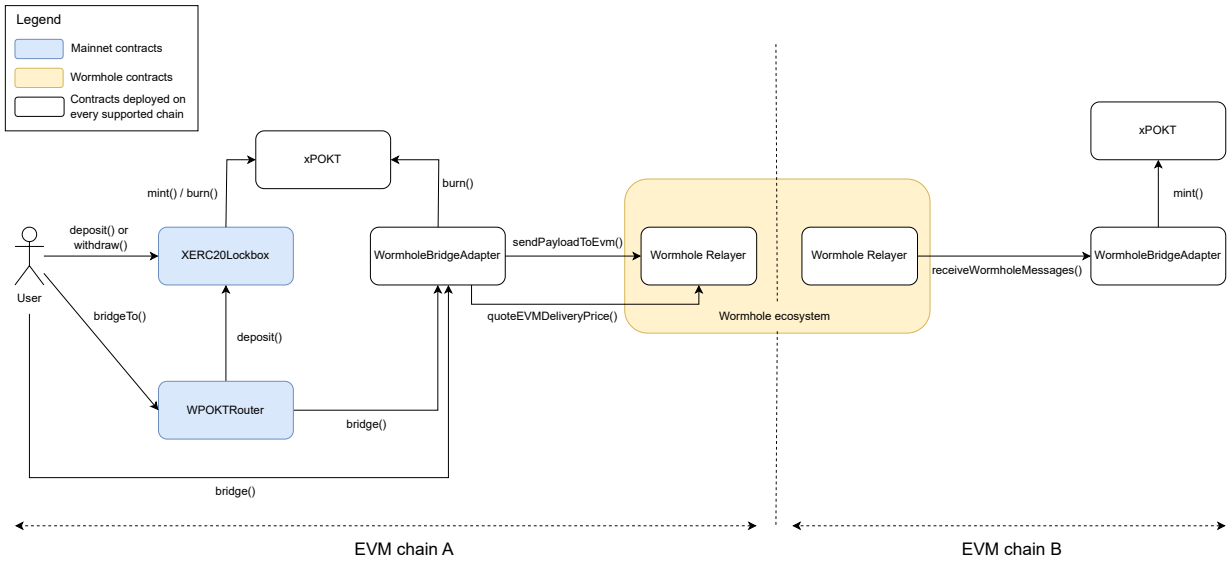


Figure 1: System overview

The following sections detail each component within the system including the `xPOKT` token, the `XERC20Lockbox` contract as well as the different flows to bridge the `xPOKT` token into other EVM chains.

4.1 xPOKT token

The `xPOKT` token is an upgradeable cross-chain token that aligns with the `xERC20` standard, designed as the native Pocket Network token that will be deployed across different EVM chains.

The token contract incorporates rate-limiting features inherited from the `xERC20` contract. This functionality allows for a controlled issuance of tokens over time by limiting the number of tokens held by each actor. Each minter is assigned a limited buffer, which is dynamically depleted or replenished with each minting or burning operation.

Moreover, the token integrates pause functionalities inherited from the `ConfigurablePauseGuardian` contract. When the token is paused, both mint and burn operations are stopped, and therefore, the token cannot be bridged.

A designated pause guardian holds the authority to halt or resume mint and burn operations via the `pause()` and `unpause()` functions:

```
function pause() public virtual whenNotPaused
function unpause() external whenPaused
```

When the contract is paused through the `pause()` function, the current timestamp is stored in the `pauseStartTime` state variable. If this pause lasts longer than a defined threshold `MAX_PAUSE_DURATION`, currently set to 30 days, the contract will automatically resume its operations as demonstrated in the implementation of the `paused()` function.

```
function paused() public view virtual override returns (bool) {
    return pauseStartTime == 0 ? false : block.timestamp <= pauseStartTime + pauseDuration;
}
```

Additionally, if the guardian fails to unpause the contract within this time window, anyone can invoke the `kickGuardian()` function to remove the guardian and resets the pausing state.

```
function kickGuardian() public whenNotPaused
```

4.2 WPOKT LockBox

Current wPOKT holders can migrate to the new xPOKT token through the XERC20Lockbox contract, exclusively deployed on the Ethereum mainnet. This contract facilitates the locking of wPOKT tokens, simultaneously minting equivalent xPOKT tokens that can be bridged later on.

Users can deposit wPOKT tokens using the deposit(...) or depositTo(...) functions, subsequently receiving the equivalent xPOKT tokens. The minted tokens are either sent to the caller or the specified address to in the depositTo(...) case.

```
function deposit(uint256 amount) external
function depositTo(address to, uint256 amount) external
```

Conversely, withdrawals are possible via the withdraw(...) and withdrawTo(...) functions. It entails burning xPOKT tokens and unlocking the corresponding wPOKT tokens, which is set back to the user.

```
function withdraw(uint256 amount) external
function withdrawTo(address to, uint256 amount) external
```

4.3 Bridging Mechanism

Bridging the xPOKT token into other chains is realized directly via the WormholeBridgeAdapter for xPOKT holders or by going through the wPOKTRouter for wPOKT holders.

4.3.1 Bridging xPOKT via WormholeBridgeAdapter

The WormholeBridgeAdapter contract serves as the interface with the Wormhole relayer, facilitating the bridging of xPOKT tokens across supported EVM chains. This contract is deployed on every chain with the same address.

The main entry point of the contract is the bridge(...) function:

```
function bridge(uint256 dstChainId, uint256 amount, address to) external payable virtual
```

This function enables users to bridge a specific amount of their xPOKT tokens to another chain by covering associated fees. The fees include a message fee for the Wormhole actor processing the message and gas fees for delivering the message. In case of message delivery failure, the message fee is refunded if the refundAddress is specified when calling sendPayloadToEvm(...).

To determine fees to be attached to the bridge(...) call, users can utilize the bridgeCost(...) function, which provides an estimation of the bridging costs for a specific destination chain dstChainId.

```
function bridgeCost(uint16 dstChainId) public view returns (uint256 gasCost)
```

Bridging the tokens via the bridge(...) function involves the following actions:

- The specified amount of xPOKT tokens is burned in the source chain.
- The sendPayloadToEvm(...) function is invoked in WormholeRelayer contract. This function sends a message to the Wormhole system, including details about the destination chain, target contract, payload data, bridging fees, and the gas limit.
- Subsequently, on the recipient chain, the Wormhole relayer initiates the receiveWormholeMessages(...) function within the WormholeBridgeAdapter. This results in the minting of xPOKT tokens for the specified receiver.

Note that the contract manages a list of trusted senders, where every WormholeBridgeAdapter contract is whitelisted, ensuring that the invocation of the receiveWormholeMessages(...) function is restricted solely to trusted senders.

4.3.2 Bridging xPOKT via wPOKTRouter

The wPOKTRouter contract serves as a utility for wPOKT holders, enabling them to directly bridge their tokens into xPOKT on a different EVM chain. This mechanism reduces the number of transactions required for the bridging process for wPOKT holders.

The main entry point in the contract is the bridgeTo(...) function:

```
function bridgeTo(uint16 chainId, address to, uint256 amount) external payable
```

The function implementation includes the following steps:

- User transfers the specified amount of wPOKT into the contract.
- These transferred tokens are then locked within the Lockbox contract through the deposit(...) function, consequently minting an equivalent amount of xPOKT tokens for the router.
- Subsequently, the bridge(...) function on the xERC20BridgeAdapter is invoked. This action results in the burning of tokens on the current EVM chain and the subsequent minting of equivalent tokens on the target chain (chainId) for the specified recipient (to).

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 [Medium] Refund address is not specified

File(s): WormholeBridgeAdapter

Description: The `WormholeBridgeAdapter` contract utilizes the Wormhole protocol to bridge tokens between different EVM chains. When users want to bridge tokens from chain A to chain B, they need to call the `bridge(...)` function, which internally triggers the `_bridgeOut(...)` function.

```
function _bridgeOut(...) internal override {
  // ...
  wormholeRelayer.sendPayloadToEvm{value: cost}(
    targetChainId,
    targetAddress[targetChainId],
    // payload
    abi.encode(to, amount),
    /// no receiver value allowed, only message passing
    0,
    gasLimit
    // @audit missing refund address and refund chain
  );
  // ...
}
```

This internal function initiates a message to the Wormhole relayer by calling `sendPayloadToEvm(...)`. The relayer processes the message and publishes it within Wormhole's infrastructure. The message is then delivered to the target address on the target chain by executing `receiveWormholeMessages(....)`. During the bridging process, the user must pay a message fee and gas, which are consumed during the `receiveWormholeMessages(...)` execution.

In the current implementation of this bridge contract, the execution of `receiveWormholeMessages(...)` can fail due to two issues:

1. The specified gas limit is too low (depending on the chain where the message is bridged);
2. RateLimiting functionality is triggered when the bridge mints an excessive number of tokens;

In instances of `receiveWormholeMessages(...)` execution failure, all paid fees associated with bridging the message, including any remaining gas, are sent to a refund address. However, since the refund address is not specified in this case, the fees will be sent to the Wormhole delivery contract and will not be returned to Pocket Network users.

Recommendation(s): To avoid losing paid fees and remaining gas after the execution of `receiveWormholeMessages(...)`, specify the `refundAddress` and `refundChain` arguments during the execution of `sendPayloadToEVM(...)`.

Status: Fixed

Update from the client: Implemented according the recommendation: 45d5841ea9cc6925aa7cbd6bd2c57ddc7a7fa7d9

6.2 [Medium] The `ownerUnpause()` function does not unpause the contract

File(s): xPOKT.sol

Description: The `xPOKT` contract incorporates a pausing mechanism by inheriting from the `ConfigurablePausableGuardian` contract. This mechanism allows designated guardians to pause the contract for a specific period, after which it automatically unpauses. The `pauseStartTime` variable is reset upon unpausing to restore the pausing capability. This reset can occur through several actions: unpausing the contract via a guardian, removing the guardian entirely, or the contract owner appointing a new guardian.

If the contract is paused, the owner can unpause it by invoking `ownerUnpause()`. However, this function is currently misimplemented, as it calls the `_unpause()` internal method inherited from `PausableUpgradable` contract of OpenZeppelin (OZ). This method resets the `paused` state variable but does not impact the `pauseStartTime`.

Recommendation(s): To address the issue, replace the call to `_unpause()` inside `ownerUnpause()` with `_resetPauseState()` internal function. This will ensure proper resetting of the `pauseStartTime` variable and removing the current guardian.

Status: Fixed

Update from the client: Implemented according the recommendation: a32f7d6c543ca3d22cd2a7c3000ff67365d61553

6.3 [Info] Allowing excessive funds in `_bridgeTo(...)` function is unnecessary

File(s): `WPOKTRouter.sol`

Description: The `_bridgeTo(...)` function facilitates the bridging of `xPOKT` tokens to a target chain. Upon invocation, users are expected to provide a value equal to or greater than the bridging cost (`bridgeCostFee`). However, only the bridging cost is forwarded to the Wormhole bridge. The excess value is always sent back to the user.

```
function _bridgeTo(uint16 chainId, address to, uint256 amount) private {
      uint256 bridgeCostFee = wormholeBridge.bridgeCost(chainId);
      //@audit User can send a value higher than needed (bridgeCostFee)
      require(
          bridgeCostFee <= msg.value,
          "WPOKTRouter: insufficient fee sent"
      );

      // ...
      // @audit Only `bridgeCostFee` is sent to the wormhole bridge
      wormholeBridge.bridge{value: bridgeCostFee}(chainId, xpoktAmount, to);

      // @audit Excess funds are sent back to the user
      if (address(this).balance != 0) {
          (bool success, ) = msg.sender.call{value: address(this).balance}("");
          require(success, "WPOKTRouter: failed to refund excess fee");
      }
      // ...
 }
```

Allowing a higher `msg.value` than the actual cost is unnecessary and leads to increased gas consumption due to the need to refund the user.

Recommendation(s): Consider modifying the condition checking `msg.value` to enforce strict equality with the bridge cost.

Status: Fixed

Update from the client: Implemented according the recommendation: 8fc47d5b978f1c424db94e340b822d44a927762f

6.4 [Info] Amount of gas can differ on various chains

File(s): `WormholeBridgeAdapter`

Description: Token bridging from chain A to chain B through the Wormhole protocol must include the payment of fees, which are calculated using `quoteEVMDeliveryPrice(...)`. This function returns the amount of ETH required for fees based on the target chain for delivery of the message and the estimated gas limit for the execution of `receiveWormholeMessages(...)`.

```
(gasCost, ) = wormholeRelayer.quoteEVMDeliveryPrice(dstChainId, 0, gasLimit);
```

The `gasLimit` is a state variable that can be modified by the contract owner, with a default value of `300_000`. However, not all chains calculate gas costs similarly to the Ethereum mainnet; thus, the required gas may vary across different chains.

Note: For example, on Arbitrum, the gas amount is calculated using the following equation: `Gas Limit (G) = Gas used on L2 (L2G) + Extra Buffer for L1 cost (B)`

If the gas amount is insufficient for the execution of `receiveWormholeMessages(...)`, the Wormhole message will need to be resent with a higher gas amount. Furthermore, due to the current usage of `sendPayloadToEvm(...)`, any remaining gas will be sent to the Wormhole delivery contract and not returned to the user.

Recommendation(s): Implement different gas limits for each chain to ensure sufficient gas is allocated for the execution of

`receiveWormholeMessages(...)`.

Status: Fixed

Update from the client: Added an option for overriding default gas limit (300_000) with a custom gas limit.

be8d7ef84b551d544b9b5e14c1a5cbae989152ff

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 Pocket Network documentation

The **Pocket Network** team has provided documentation about their protocol based on the following HackMD. Additionally, the Pocket Network team was available to address any questions or concerns from the Nethermind Security team.

8 Test Suite Evaluation

8.1 Compilation Output

```
> forge compile
[] Compiling...
[] Compiling 9 files with Solc 0.8.19
[] Solc 0.8.19 finished in 762.24ms
Compiler run successful!
```

8.2 Tests Output

```
> forge test

Ran 9 tests for test/unit/xPOKTPause.t.sol:xPOKTPauseUnitTest
[PASS] testCanKickGuardianAfterPauseUsed() (gas: 59131)
[PASS] testGuardianCanPause() (gas: 39841)
[PASS] testGuardianCanUnpause() (gas: 54285)
[PASS] testKickFailsWithZeroStartTime() (gas: 19634)
[PASS] testKickGuardianSucceedsAfterUnpause() (gas: 57130)
[PASS] testPauseFailsPauseAlreadyUsed() (gas: 53220)
[PASS] testPauseNonGuardianFails() (gas: 21088)
[PASS] testShouldUnpauseAutomaticallyAfterPauseDuration() (gas: 49483)
[PASS] testUnpauseNonGuardianFails() (gas: 43278)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 25.86ms (3.72ms CPU time)

Ran 25 tests for test/unit/WormholeBridgeAdapter.t.sol:WormholeBridgeAdapterUnitTest
[PASS] testAddTrustedSendersNonOwnerFails() (gas: 19668)
[PASS] testAddTrustedSendersOwnerFailsAlreadyWhitelisted(address) (runs: 1, : 86279, ~: 86279)
[PASS] testAddTrustedSendersOwnerSucceeds(address) (runs: 1, : 80501, ~: 80501)
[PASS] testAllTrustedSendersTrusted() (gas: 28977)
[PASS] testAlreadyProcessedMessageReplayFails(bytes32) (runs: 1, : 153904, ~: 153904)
[PASS] testBridgeInFailsRateLimitExhausted(bytes32) (runs: 1, : 189618, ~: 189618)
[PASS] testBridgeOutFailsIncorrectCost() (gas: 37292)
[PASS] testBridgeOutFailsIncorrectTargetChain() (gas: 32352)
[PASS] testBridgeOutFailsNoApproval() (gas: 64026)
[PASS] testBridgeOutFailsNotEnoughBalance() (gas: 306504)
[PASS] testBridgeOutFailsNotEnoughBuffer() (gas: 200768)
[PASS] testBridgeOutSucceeds() (gas: 321592)
[PASS] testInitializingFails() (gas: 26863)
[PASS] testReceiveWormholeMessageFailsNotRelayer() (gas: 22069)
[PASS] testReceiveWormholeMessageFailsNotTrustedExternalChain() (gas: 26753)
[PASS] testReceiveWormholeMessageFailsWithValue() (gas: 27714)
[PASS] testReceiveWormholeMessageSucceeds(bytes32) (runs: 1, : 147777, ~: 147777)
[PASS] testRemoveNonTrustedSendersOwnerFails() (gas: 71802)
[PASS] testRemoveTrustedSendersNonOwnerFails() (gas: 19668)
[PASS] testRemoveTrustedSendersOwnerSucceeds() (gas: 68512)
[PASS] testSetGasLimitNonOwnerFails() (gas: 18773)
[PASS] testSetGasLimitOwnerSucceeds(uint96) (runs: 1, : 34232, ~: 34232)
[PASS] testSetTargetAddressesNonOwnerFails() (gas: 19030)
[PASS] testSetTargetAddressesOwnerSucceeds(address,uint16) (runs: 1, : 51245, ~: 51245)
[PASS] testSetup() (gas: 83806)
Suite result: ok. 25 passed; 0 failed; 0 skipped; finished in 25.98ms (32.46ms CPU time)
```

```
Ran 55 tests for test/unit/xPOKT.t.sol:xPOKTUnitTest
[PASS] testAddBridgeNonOwnerReverts() (gas: 46549)
[PASS] testAddBridgesNonOwnerReverts() (gas: 47700)
[PASS] testAddNewBridgeBufferCapZeroFails() (gas: 21930)
[PASS] testAddNewBridgeInvalidAddressFails() (gas: 23428)
[PASS] testAddNewBridgeOverMaxRateLimitPerSecondFails() (gas: 22281)
[PASS] testAddNewBridgeOwnerSucceeds(address,uint128,uint112) (runs: 1, : 92787, ~: 92787)
[PASS] testAddNewBridgeWithBufferBelowMinFails() (gas: 26453)
[PASS] testAddNewBridgeWithExistingLimitFails() (gas: 96758)
[PASS] testAddNewBridgesOwnerSucceeds(address,uint128,uint112) (runs: 1, : 93588, ~: 93588)
[PASS] testBurnFailsWhenPaused() (gas: 37790)
[PASS] testCannotRemoveNonExistentBridge() (gas: 20719)
[PASS] testCannotRemoveNonExistentBridges() (gas: 22822)
[PASS] testDecreaseAllowance(uint256) (runs: 1, : 43784, ~: 43784)
[PASS] testDepleteBufferBridgeByZeroFails() (gas: 98750)
[PASS] testDepleteBufferBridgeSucceeds() (gas: 173060)
[PASS] testDepleteBufferNonBridgeByOneFails() (gas: 28815)
[PASS] testGrantGuardianNonOwnerReverts() (gas: 45105)
[PASS] testGrantGuardianOwnerSucceeds(address) (runs: 1, : 31047, ~: 31047)
[PASS] testGrantPauseGuardianWhilePausedFails() (gas: 41058)
[PASS] testIncreaseAllowance(uint256) (runs: 1, : 49520, ~: 49520)
[PASS] testInitializationFailsPauseDurationGtMax() (gas: 117102)
[PASS] testInitializeLogicContractFails() (gas: 25949)
[PASS] testLockBoxCanBurn(uint112) (runs: 1, : 226604, ~: 226604)
[PASS] testLockBoxCanMintBurn(uint112) (runs: 1, : 228548, ~: 228548)
[PASS] testLockboxCanMint(uint112) (runs: 1, : 167545, ~: 167545)
[PASS] testLockboxCanMintBurnTo(uint112) (runs: 1, : 226984, ~: 226984)
[PASS] testLockboxCanMintTo(address,uint112) (runs: 1, : 167876, ~: 167876)
[PASS] testMintFailsWhenPaused() (gas: 36646)
[PASS] testMintSucceedsAfterPauseDuration() (gas: 192131)
[PASS] testNonOwnerUnpauseFails() (gas: 19071)
[PASS] testOwnerCanUnpause() (gas: 43729)
[PASS] testOwnerUnpauseFailsNotPaused() (gas: 21985)
[PASS] testPendingOwnerAccepts() (gas: 43904)
[PASS] testPermit(uint256) (runs: 1, : 93845, ~: 93845)
[PASS] testRemoveBridgeNonOwnerReverts() (gas: 45985)
[PASS] testRemoveBridgeOwnerSucceeds() (gas: 37990)
[PASS] testRemoveBridgesNonOwnerReverts() (gas: 45848)
[PASS] testRemoveBridgesOwnerSucceeds() (gas: 92305)
[PASS] testReplenishBufferBridgeByZeroFails() (gas: 98662)
[PASS] testReplenishBufferBridgeSucceeds() (gas: 169867)
[PASS] testReplenishBufferNonBridgeByOneFails() (gas: 28861)
[PASS] testSetBridgeBufferBelowMinFails() (gas: 98981)
[PASS] testSetBufferCapNonOwnerReverts() (gas: 46547)
[PASS] testSetBufferCapOnNonExistentBridgeFails(uint112) (runs: 1, : 22695, ~: 22695)
[PASS] testSetBufferCapOwnerSucceeds(uint112) (runs: 1, : 44834, ~: 44834)
[PASS] testSetBufferCapZeroFails() (gas: 21763)
[PASS] testSetExistingBridgeOverMaxRateLimitPerSecondFails() (gas: 25376)
[PASS] testSetPauseDurationNonOwnerReverts() (gas: 46126)
[PASS] testSetRateLimitOnNonExistentBridgeFails(uint128) (runs: 1, : 25437, ~: 25437)
[PASS] testSetRateLimitPerSecondNonOwnerReverts() (gas: 46960)
[PASS] testSetRateLimitPerSecondOwnerSucceeds(uint128) (runs: 1, : 39443, ~: 39443)
[PASS] testSetup() (gas: 164286)
[PASS] testTransferToTokenContractFails() (gas: 170181)
[PASS] testUpdatePauseDurationGtMaxPauseDurationFails() (gas: 21177)
[PASS] testUpdatePauseDurationSucceeds() (gas: 28995)
Suite result: ok. 55 passed; 0 failed; 0 skipped; finished in 26.66ms (63.35ms CPU time)

Ran 3 test suites in 64.50ms (78.51ms CPU time): 89 tests passed, 0 failed, 0 skipped (89 total tests)
```

Remarks about Pocket Network test suite

The **Pocket Network** team has developed a comprehensive test suite encompassing various user workflows and integration scenarios with the Wormhole protocol. This rigorous testing has contributed to a well-written and secure codebase. However, the integration tests only utilized Wormhole mock contracts, which limited the testing of real integration behavior and the completeness of Wormhole message delivery failures.

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.