Wormhole Foundation EVM-NTT Audit Report

Prepared by Cyfrin

Version 1.1

Lead Auditors

0kage

Giovanni Di Siena

Assisting Auditors

Hans

March 28, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

4.1 Wormhole Native Token Transfers (NTT)

Wormhole is a generic message passing protocol that enables communication between blockchains. NTT is a framework designed to facilitate the transfer of tokens across different blockchain networks without relying on liquidity pools, offering an open, flexible, and composable solution.

The Wormhole NTT framework grants integrators complete control over the behavior of NTTs on each chain, including the token standard and metadata. It supports a "locking" mode for existing token deployments to preserve the original token supply on a single chain, and a "burning" mode for deploying natively multichain tokens with a distributed supply.

4.1.1 Key Components:

- **Transceivers:** Responsible for sending NTT transfers, managed by the `NttManager` contract on the source chain and delivered to the corresponding `NttManager` peer on the recipient chain.
- **NttManager**: Manages tokens and Transceivers, including rate-limiting and message attestation logic.

4.1.2 Design Features:

- Provision for rate-limiting transfers on both source and destination chains, with excess transfers queued for delayed execution.
- Users must pay a gas fee sufficient for all Transceivers to handle transfers on target chains.
- NTT Managers can use Wormhole transceivers or implement customized versions.

4.1.3 Architecture

The architecture of this framework primarily consists of the `NttManager` and `Transceiver` contracts (of which `WormholeTransceiver` is a specific implementation), which interface with the external Wormhole Core Relayer and Bridge contracts. A summary of the architecture, including important function calls and their return values, is shown below:

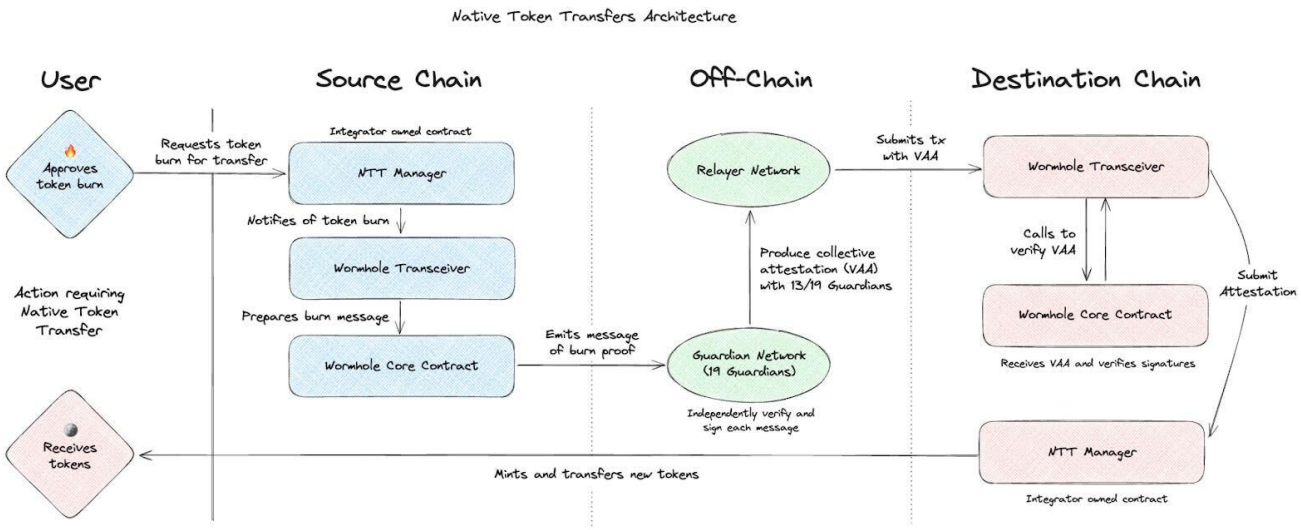

Figure 1: Wormhole Native Token Transfers Architecture

4.1.4 Threat Model

NTT Manager API

Transceiver::sendMessage

- Digest collision
- Zero-value transfer
- Rate limiter bypassed for invalid chain ID
- Relayer abuse

Governance API

Governance::performGovernance

- Replayed VAA
- Fraudulent VAA
- Cross-chain replay
- Invalid message/chain/source/action
- Impersonation of governance module
- Execution for invalid target chain
- Execution of low-level call hijacked (e.g. returndata bomb)

Transceiver (Relayer) API

WormholeTransciever::receiveMessage

- Receive invalid message

User API

NttManager::transfer

- Amount mismatch
- Invalid recipient chain
- Bad (e.g. blacklisted) recipient
- Non-standard token (e.g. fees, callback, etc)
- Abuse queue
- Invalid transceiver instructions
- Mode asymmetries between source & destination
- Token asymmetries between source & destination

NttManager::completeInboundQueuedTransfer

- Complete invalid transfer
- Replay

NttManager::completeOutboundQueuedTransfer

- Complete invalid transfer
- Replay

NttManager::executeMsg

- Execute invalid message
- Invalid source chain
- Invalid source chain manager address
- Replay

5 Audit Scope

Cyfrin conducted an audit of the Wormhole Native Token Transfers (NTT) based on the code present in the repository commit hash f4e2277.

The following directories were included in the scope of the audit:

- evm/src/NttManager/*
- evm/src/Transceiver/*
- evm/src/interfaces/*
- evm/src/libraries/*
- evm/src/wormhole/*

6 Executive Summary

Over the course of 20 days, the Cyfrin team conducted an audit on the Wormhole Foundation EVM-NTT smart contracts provided by Wormhole Foundation. In this period, a total of 25 issues were found.

This review of the Wormhole Native Token Transfers (NTT) contracts yielded two scenarios that could result in user funds becoming stuck, where the completion of queued transfers on the source chain could revert due to non-sequential Transceiver instructions and modifications to the registered Transceivers during the rate-limiting period. A number of medium-severity findings were raised relating to gas usage, Transceiver modifications, DoS of admin functionality, and the silent overflow of transfer amounts. A number of additional low, informational, and gas-optimization findings have been raised, where there is no immediate impact but potential for improvement to help prevent issues from occurring in future.

The inline documentation of the codebase was deemed sufficient, though some areas, particularly those related to Transceiver operations, could benefit from more detailed explanations. Instances where there is responsibility on integrators to correctly handle certain behaviors should be clearly documented and communicated. The test suite coverage was satisfactory, encompassing both unit and integration tests covering all core functionalities of the codebase. Overall, this is a well-architected and well-tested system with a strong focus on security and reliability.

The audit process was somewhat complicated by parallel code updates, refactoring, and bug fixes, necessitating ongoing communication with the development team and a number of changes to the final audit commit hash. Cyfrin recommends addressing the high and medium risk issues as a priority to ensure the security and reliability of the Wormhole NTT framework.

Summary

Project Name	Wormhole Foundation EVM-NTT
Repository	example-native-token-transfers
Commit	f4e2277b3583...
Audit Timeline	Feb 26th - Mar 22nd
Methods	Manual Review, Fuzzing

Issues Found

Critical Risk	0
High Risk	2
Medium Risk	8
Low Risk	5
Informational	6
Gas Optimizations	4
Total Issues	25

Summary of Findings

[H-1] Queued transfers can become stuck on the source chain if Transceiver instructions are encoded in the incorrect order	Open
[H-2] Queued transfers can become stuck on the source chain if new Transceivers are added or existing Transceivers are modified before completion	Open
[M-1] Silent overflow in `TrimmedAmount::shift` could result in rate limiter being bypassed	Open
[M-2] Disabled Transceivers cannot be re-enabled by calling `TransceiverRegistry::_setTransceiver` after 64 have been registered	Open
[M-3] NTT Manager cannot be unpaused once paused	Open
[M-4] Immutable gas limit within `WormholeTransceiver` can lead to execution failures on the target chain	Open
[M-5] Transceiver invariants and ownership synchronicity can be broken by unsafe Transceiver upgrades	Open
[M-6] Setting a peer `NttManager` contract for the same chain can cause loss of user funds	Open
[M-7] Lack of a gas refund in the current design can lead to the overcharging of users and misaligned relayer incentives that can choke message execution	Open
[M-8] Access-controlled functions cannot be called when L2 sequencers are down	Open
[L-1] Inconsistent implementation of ERC-7201 Namespaced Storage locations	Open
[L-2] Asymmetry in Transceiver pausing capability	Open
[L-3] Incorrect Transceiver payload prefix definition	Open
[L-4] `WormholeTransceiver` EVM chain IDs storage cannot be updated	Open
[L-5] Missing threshold invariant check when adding/removing transceivers	Open
[I-1] Unchained initializers should be called in `PausableOwnable::__PausedOwnable_init` instead	Open
[I-2] Incorrect topics[0] documented in `INTTManagerEvents` and `IRateLimiterEvents`	Open
[I-3] NatSpec documentation inconsistent with function/event signatures	Open
[I-4] Unused `PausableUpgradeable::CannotRenounceWhilePaused` error should be removed	Open
[I-5] Multiple `staticcall` success booleans are not checked	Open
[I-6] Canonical NTT chain ID should be fetched directly from Wormhole or mapped accordingly	Open
[G-1] Unnecessary type casts of proxied storage slots from `bytes32` to `uint256`	Open
[G-2] Unnecessary stack variable in `Governance::encodeGeneralPurposeGovernanceMessage`	Open
[G-3] Optimize the `TrimmedAmount` struct as a bit-packed `uint72` user-defined type	Open

[G-4] Optimize invariant check iterations in `TransceiverRegistry::_checkTransceiversInvariants`	Open

7 Findings

7.1 High Risk

7.1.1 Queued transfers can become stuck on the source chain if Transceiver instructions are encoded in the incorrect order

Description: In the case of multiple Transceivers, the current logic expects that a sender encodes Transceiver instructions in order of increasing Transceiver registration index, as validated in `TransceiverStructs::parseTransceiverInstructions`. Under normal circumstances, this logic works as expected, and the transaction fails when the user packs transceiver instructions in the incorrect order.

```
/* snip */
for (uint256 i = 0; i < instructionsLength; i++) {
    TransceiverInstruction memory instruction;
    (instruction, offset) = parseTransceiverInstructionUnchecked(encoded, offset);

    uint8 instructionIndex = instruction.index;

    // The instructions passed in have to be strictly increasing in terms of transceiver index
    if (i != 0 && instructionIndex <= lastIndex) {
        revert UnorderedInstructions();
    }
    lastIndex = instructionIndex;

    instructions[instructionIndex] = instruction;
}
/* snip */
```

However, this requirement on the order of Transceiver indices is not checked when transfers are initially queued for delayed execution. As a result, a transaction where this is the case will fail when the user calls `NttManager::completeOutboundQueuedTransfer` to execute a queued transfer.

Impact: The sender's funds are transferred to the NTT Manager when messages are queued. However, this queued message can never be executed if the Transceiver indices are incorrectly ordered and, as a result, the user funds remain stuck in the NTT Manager.

Proof of Concept: Run the following test:

```
contract TestWrongTransceiverOrder is Test, INttManagerEvents, IRateLimiterEvents {
    NttManager nttManagerChain1;
    NttManager nttManagerChain2;

    using TrimmedAmountLib for uint256;
    using TrimmedAmountLib for TrimmedAmount;

    uint16 constant chainId1 = 7;
    uint16 constant chainId2 = 100;
    uint8 constant FAST_CONSISTENCY_LEVEL = 200;
    uint256 constant GAS_LIMIT = 500000;

    uint16 constant SENDING_CHAIN_ID = 1;
    uint256 constant DEVNET_GUARDIAN_PK =
        0xcfb12303a19cde580bb4dd771639b0d26bc68353645571a8cff516ab2ee113a0;
    WormholeSimulator guardian;
    uint256 initialBlockTimestamp;

    WormholeTransceiver wormholeTransceiverChain1;
    WormholeTransceiver wormholeTransceiver2Chain1;

    WormholeTransceiver wormholeTransceiverChain2;
```

```
address userA = address(0x123);
address userB = address(0x456);
address userC = address(0x789);
address userD = address(0xABC);

address relayer = address(0x28D8F1Be96f97C1387e94A53e00eCcFb4E75175a);
IWormhole wormhole = IWormhole(0x706abc4E45D419950511e474C7B9Ed348A4a716c);

function setUp() public {
    string memory url = "https://goerli.blockpi.network/v1/rpc/public";
    vm.createSelectFork(url);
    initialBlockTimestamp = vm.getBlockTimestamp();

    guardian = new WormholeSimulator(address(wormhole), DEVNET_GUARDIAN_PK);

    vm.chainId(chainId1);
    DummyToken t1 = new DummyToken();
    NttManager implementation =
        new MockNttManagerContract(address(t1), INttManager.Mode.LOCKING, chainId1, 1 days);

    nttManagerChain1 =
        MockNttManagerContract(address(new ERC1967Proxy(address(implementation), "")));
    nttManagerChain1.initialize();

    WormholeTransceiver wormholeTransceiverChain1Implementation = new
    ↪  MockWormholeTransceiverContract(
        address(nttManagerChain1),
        address(wormhole),
        address(relayer),
        address(0x0),
        FAST_CONSISTENCY_LEVEL,
        GAS_LIMIT
    );
    wormholeTransceiverChain1 = MockWormholeTransceiverContract(
        address(new ERC1967Proxy(address(wormholeTransceiverChain1Implementation), ""))
    );

    WormholeTransceiver wormholeTransceiverChain1Implementation2 = new
    ↪  MockWormholeTransceiverContract(
        address(nttManagerChain1),
        address(wormhole),
        address(relayer),
        address(0x0),
        FAST_CONSISTENCY_LEVEL,
        GAS_LIMIT
    );
    wormholeTransceiver2Chain1 = MockWormholeTransceiverContract(
        address(new ERC1967Proxy(address(wormholeTransceiverChain1Implementation2), ""))
    );


    // Actually initialize properly now
    wormholeTransceiverChain1.initialize();
    wormholeTransceiver2Chain1.initialize();


    nttManagerChain1.setTransceiver(address(wormholeTransceiverChain1));
    nttManagerChain1.setTransceiver(address(wormholeTransceiver2Chain1));
    nttManagerChain1.setOutboundLimit(type(uint64).max);
    nttManagerChain1.setInboundLimit(type(uint64).max, chainId2);

    // Chain 2 setup
```

```
        vm.chainId(chainId2);
        DummyToken t2 = new DummyTokenMintAndBurn();
        NttManager implementationChain2 =
            new MockNttManagerContract(address(t2), INttManager.Mode.BURNING, chainId2, 1 days);

        nttManagerChain2 =
            MockNttManagerContract(address(new ERC1967Proxy(address(implementationChain2), "")));
        nttManagerChain2.initialize();

        WormholeTransceiver wormholeTransceiverChain2Implementation = new
        ↪  MockWormholeTransceiverContract(
            address(nttManagerChain2),
            address(wormhole),
            address(relayer),
            address(0x0),
            FAST_CONSISTENCY_LEVEL,
            GAS_LIMIT
        );

        wormholeTransceiverChain2 = MockWormholeTransceiverContract(
            address(new ERC1967Proxy(address(wormholeTransceiverChain2Implementation), ""))
        );
        wormholeTransceiverChain2.initialize();

        nttManagerChain2.setTransceiver(address(wormholeTransceiverChain2));
        nttManagerChain2.setOutboundLimit(type(uint64).max);
        nttManagerChain2.setInboundLimit(type(uint64).max, chainId1);

        // Register peer contracts for the nttManager and transceiver. Transceivers and nttManager each
        ↪  have the concept of peers here.
        nttManagerChain1.setPeer(chainId2, bytes32(uint256(uint160(address(nttManagerChain2)))), 9);
        nttManagerChain2.setPeer(chainId1, bytes32(uint256(uint160(address(nttManagerChain1)))), 7);

        // Set peers for the transceivers
        wormholeTransceiverChain1.setWormholePeer(
            chainId2, bytes32(uint256(uint160(address(wormholeTransceiverChain2))))
        );

       wormholeTransceiver2Chain1.setWormholePeer(
            chainId2, bytes32(uint256(uint160(address(wormholeTransceiverChain2))))
        );

        wormholeTransceiverChain2.setWormholePeer(
            chainId1, bytes32(uint256(uint160(address(wormholeTransceiverChain1))))
        );

        require(nttManagerChain1.getThreshold() != 0, "Threshold is zero with active transceivers");

        // Actually set it
        nttManagerChain1.setThreshold(2);
        nttManagerChain2.setThreshold(1);
    }

    function testWrongTransceiverOrder() external {
        vm.chainId(chainId1);

        // Setting up the transfer
        DummyToken token1 = DummyToken(nttManagerChain1.token());
        uint8 decimals = token1.decimals();

        token1.mintDummy(address(userA), 5 * 10 ** decimals);
        uint256 outboundLimit = 4 * 10 ** decimals;
```

```
        nttManagerChain1.setOutboundLimit(outboundLimit);

        vm.startPrank(userA);

        uint256 transferAmount = 5 * 10 ** decimals;
        token1.approve(address(nttManagerChain1), transferAmount);

        // transfer with shouldQueue == true
        uint64 qSeq = nttManagerChain1.transfer(
            transferAmount, chainId2, toWormholeFormat(userB), true,
            ↪  encodeTransceiverInstructionsJumbled(true)
        );

        assertEq(qSeq, 0);
        IRateLimiter.OutboundQueuedTransfer memory qt = nttManagerChain1.getOutboundQueuedTransfer(0);
        assertEq(qt.amount.getAmount(), transferAmount.trim(decimals, decimals).getAmount());
        assertEq(qt.recipientChain, chainId2);
        assertEq(qt.recipient, toWormholeFormat(userB));
        assertEq(qt.txTimestamp, initialBlockTimestamp);

        // assert that the contract also locked funds from the user
        assertEq(token1.balanceOf(address(userA)), 0);
        assertEq(token1.balanceOf(address(nttManagerChain1)), transferAmount);

         // elapse rate limit duration - 1
        uint256 durationElapsedTime = initialBlockTimestamp + nttManagerChain1.rateLimitDuration();

        vm.warp(durationElapsedTime);

        vm.expectRevert(0x71f23ef2); //UnorderedInstructions() selector
        nttManagerChain1.completeOutboundQueuedTransfer(0);
    }

    // Encode an instruction for each of the relayers
    function encodeTransceiverInstructionsJumbled(bool relayer_off) public view returns (bytes memory) {
        WormholeTransceiver.WormholeTransceiverInstruction memory instruction =
            IWormholeTransceiver.WormholeTransceiverInstruction(relayer_off);

        bytes memory encodedInstructionWormhole =
            wormholeTransceiverChain1.encodeWormholeTransceiverInstruction(instruction);

        TransceiverStructs.TransceiverInstruction memory TransceiverInstruction1 =
        TransceiverStructs.TransceiverInstruction({index: 0, payload: encodedInstructionWormhole});
        TransceiverStructs.TransceiverInstruction memory TransceiverInstruction2 =
        TransceiverStructs.TransceiverInstruction({index: 1, payload: encodedInstructionWormhole});

        TransceiverStructs.TransceiverInstruction[] memory TransceiverInstructions =
            new TransceiverStructs.TransceiverInstruction[](2);

        TransceiverInstructions[0] = TransceiverInstruction2;
        TransceiverInstructions[1] = TransceiverInstruction1;

        return TransceiverStructs.encodeTransceiverInstructions(TransceiverInstructions);
    }
}
```

Recommended Mitigation: When the transfer amount exceeds the current outbound capacity, verify the Transceiver instructions are ordered correctly before adding a message to the list of queued transfers.

7.1.2 Queued transfers can become stuck on the source chain if new Transceivers are added or existing Transceivers are modified before completion

Description: When a sender transfers an amount that exceeds the current outbound capacity, such transfers are sent to a queue for delayed execution within `NttManager::_transferEntrypoint`. The rate limit duration is defined as an immutable variable determining the temporal lag between queueing and execution, with a typical rate limit duration being 24 hours.

```
/* snip */
// now check rate limits
bool isAmountRateLimited = _isOutboundAmountRateLimited(internalAmount);
if (!shouldQueue && isAmountRateLimited) {
    revert NotEnoughCapacity(getCurrentOutboundCapacity(), amount);
}
if (shouldQueue && isAmountRateLimited) {
    // emit an event to notify the user that the transfer is rate limited
    emit OutboundTransferRateLimited(
        msg.sender, sequence, amount, getCurrentOutboundCapacity()
    );

    // queue up and return
    _enqueueOutboundTransfer(
        sequence,
        trimmedAmount,
        recipientChain,
        recipient,
        msg.sender,
        transceiverInstructions
    );

    // refund price quote back to sender
    _refundToSender(msg.value);

    // return the sequence in the queue
    return sequence;
}
/* snip */
```

In the event that new Transceivers are added or existing Transceivers are removed from the NTT Manager, any pending queued transfers within the rate limit duration can potentially revert. This is because senders might not have correctly packed the Transceiver instructions for a given Transceiver based on the new configuration, and a missing Transceiver instruction can potentially cause an array index out-of-bounds exception while calculating the delivery price when the instructions are finally parsed. For example, if there are initially two Transceivers but an additional Transceiver is added while the transfer is rate-limited, the instructions array as shown below will be declared with a length of three, corresponding to the new number of enabled Transceivers; however, the transfer will have only encoded two Transceiver instructions based on the configuration at the time it was initiated.

```
function parseTransceiverInstructions(
    bytes memory encoded,
    uint256 numEnabledTransceivers
) public pure returns (TransceiverInstruction[] memory) {
    uint256 offset = 0;
    uint256 instructionsLength;
    (instructionsLength, offset) = encoded.asUint8Unchecked(offset);

    // We allocate an array with the length of the number of enabled transceivers
    // This gives us the flexibility to not have to pass instructions for transceivers that
    // don't need them
    TransceiverInstruction[] memory instructions =
        new TransceiverInstruction[](numEnabledTransceivers);
```

```
    uint256 lastIndex = 0;
    for (uint256 i = 0; i < instructionsLength; i++) {
        TransceiverInstruction memory instruction;
        (instruction, offset) = parseTransceiverInstructionUnchecked(encoded, offset);

        uint8 instructionIndex = instruction.index;

        // The instructions passed in have to be strictly increasing in terms of transceiver index
        if (i != 0 && instructionIndex <= lastIndex) {
            revert UnorderedInstructions();
        }
        lastIndex = instructionIndex;

        instructions[instructionIndex] = instruction;
    }

    encoded.checkLength(offset);

    return instructions;
}
```

Impact: Missing Transceiver instructions prevents the total delivery price for the corresponding message from being calculated. This prevents any queued Transfers from being executed with the current list of transceivers. As a result, underlying sender funds will be stuck in the `NttManager` contract. Note that a similar issue occurs if the peer NTT manager contract is updated on the destination (say, after a redeployment on the source chain) before an in-flight attestation is received and executed, reverting with an invalid peer error.

Proof of Concept: Run the following test:

```
contract TestTransceiverModification is Test, INttManagerEvents, IRateLimiterEvents {
    NttManager nttManagerChain1;
    NttManager nttManagerChain2;

    using TrimmedAmountLib for uint256;
    using TrimmedAmountLib for TrimmedAmount;

    uint16 constant chainId1 = 7;
    uint16 constant chainId2 = 100;
    uint8 constant FAST_CONSISTENCY_LEVEL = 200;
    uint256 constant GAS_LIMIT = 500000;

    uint16 constant SENDING_CHAIN_ID = 1;
    uint256 constant DEVNET_GUARDIAN_PK =
        0xcfb12303a19cde580bb4dd771639b0d26bc68353645571a8cff516ab2ee113a0;
    WormholeSimulator guardian;
    uint256 initialBlockTimestamp;

    WormholeTransceiver wormholeTransceiverChain1;
    WormholeTransceiver wormholeTransceiver2Chain1;
    WormholeTransceiver wormholeTransceiver3Chain1;

    WormholeTransceiver wormholeTransceiverChain2;
    address userA = address(0x123);
    address userB = address(0x456);
    address userC = address(0x789);
    address userD = address(0xABC);

    address relayer = address(0x28D8F1Be96f97C1387e94A53e00eCcFb4E75175a);
    IWormhole wormhole = IWormhole(0x706abc4E45D419950511e474C7B9Ed348A4a716c);
```

```
function setUp() public {
    string memory url = "https://goerli.blockpi.network/v1/rpc/public";
    vm.createSelectFork(url);
    initialBlockTimestamp = vm.getBlockTimestamp();

    guardian = new WormholeSimulator(address(wormhole), DEVNET_GUARDIAN_PK);

    vm.chainId(chainId1);
    DummyToken t1 = new DummyToken();
    NttManager implementation =
        new MockNttManagerContract(address(t1), INttManager.Mode.LOCKING, chainId1, 1 days);

    nttManagerChain1 =
        MockNttManagerContract(address(new ERC1967Proxy(address(implementation), "")));
    nttManagerChain1.initialize();

    // transceiver 1
    WormholeTransceiver wormholeTransceiverChain1Implementation = new
    ↪  MockWormholeTransceiverContract(
        address(nttManagerChain1),
        address(wormhole),
        address(relayer),
        address(0x0),
        FAST_CONSISTENCY_LEVEL,
        GAS_LIMIT
    );
    wormholeTransceiverChain1 = MockWormholeTransceiverContract(
        address(new ERC1967Proxy(address(wormholeTransceiverChain1Implementation), ""))
    );

    // transceiver 2
    WormholeTransceiver wormholeTransceiverChain1Implementation2 = new
    ↪  MockWormholeTransceiverContract(
        address(nttManagerChain1),
        address(wormhole),
        address(relayer),
        address(0x0),
        FAST_CONSISTENCY_LEVEL,
        GAS_LIMIT
    );
    wormholeTransceiver2Chain1 = MockWormholeTransceiverContract(
        address(new ERC1967Proxy(address(wormholeTransceiverChain1Implementation2), ""))
    );

    // transceiver 3
    WormholeTransceiver wormholeTransceiverChain1Implementation3 = new
    ↪  MockWormholeTransceiverContract(
        address(nttManagerChain1),
        address(wormhole),
        address(relayer),
        address(0x0),
        FAST_CONSISTENCY_LEVEL,
        GAS_LIMIT
    );
    wormholeTransceiver3Chain1 = MockWormholeTransceiverContract(
        address(new ERC1967Proxy(address(wormholeTransceiverChain1Implementation3), ""))
    );


    // Actually initialize properly now
    wormholeTransceiverChain1.initialize();
    wormholeTransceiver2Chain1.initialize();
```

```
wormholeTransceiver3Chain1.initialize();


nttManagerChain1.setTransceiver(address(wormholeTransceiverChain1));
nttManagerChain1.setTransceiver(address(wormholeTransceiver2Chain1));

// third transceiver is NOT set at this point for nttManagerChain1
nttManagerChain1.setOutboundLimit(type(uint64).max);
nttManagerChain1.setInboundLimit(type(uint64).max, chainId2);

// Chain 2 setup
vm.chainId(chainId2);
DummyToken t2 = new DummyTokenMintAndBurn();
NttManager implementationChain2 =
    new MockNttManagerContract(address(t2), INttManager.Mode.BURNING, chainId2, 1 days);

nttManagerChain2 =
    MockNttManagerContract(address(new ERC1967Proxy(address(implementationChain2), "")));
nttManagerChain2.initialize();

WormholeTransceiver wormholeTransceiverChain2Implementation = new
↪  MockWormholeTransceiverContract(
    address(nttManagerChain2),
    address(wormhole),
    address(relayer),
    address(0x0),
    FAST_CONSISTENCY_LEVEL,
    GAS_LIMIT
);

wormholeTransceiverChain2 = MockWormholeTransceiverContract(
    address(new ERC1967Proxy(address(wormholeTransceiverChain2Implementation), ""))
);
wormholeTransceiverChain2.initialize();

nttManagerChain2.setTransceiver(address(wormholeTransceiverChain2));
nttManagerChain2.setOutboundLimit(type(uint64).max);
nttManagerChain2.setInboundLimit(type(uint64).max, chainId1);

// Register peer contracts for the nttManager and transceiver. Transceivers and nttManager each
↪  have the concept of peers here.
nttManagerChain1.setPeer(chainId2, bytes32(uint256(uint160(address(nttManagerChain2)))), 9);
nttManagerChain2.setPeer(chainId1, bytes32(uint256(uint160(address(nttManagerChain1)))), 7);

// Set peers for the transceivers
wormholeTransceiverChain1.setWormholePeer(
    chainId2, bytes32(uint256(uint160(address(wormholeTransceiverChain2))))
);

wormholeTransceiver2Chain1.setWormholePeer(
    chainId2, bytes32(uint256(uint160(address(wormholeTransceiverChain2))))
);

wormholeTransceiver3Chain1.setWormholePeer(
    chainId2, bytes32(uint256(uint160(address(wormholeTransceiverChain2))))
);


wormholeTransceiverChain2.setWormholePeer(
    chainId1, bytes32(uint256(uint160(address(wormholeTransceiverChain1))))
);
```

```
        require(nttManagerChain1.getThreshold() != 0, "Threshold is zero with active transceivers");

        // Actually set it
        nttManagerChain1.setThreshold(2);
        nttManagerChain2.setThreshold(1);
    }

    function testTransceiverModification() external {
        vm.chainId(chainId1);

        // Setting up the transfer
        DummyToken token1 = DummyToken(nttManagerChain1.token());
        uint8 decimals = token1.decimals();

        token1.mintDummy(address(userA), 5 * 10 ** decimals);
        uint256 outboundLimit = 4 * 10 ** decimals;
        nttManagerChain1.setOutboundLimit(outboundLimit);

        vm.startPrank(userA);

        uint256 transferAmount = 5 * 10 ** decimals;
        token1.approve(address(nttManagerChain1), transferAmount);

        // transfer with shouldQueue == true
        uint64 qSeq = nttManagerChain1.transfer(
            transferAmount, chainId2, toWormholeFormat(userB), true, encodeTransceiverInstructions(true)
        );
        vm.stopPrank();

        assertEq(qSeq, 0);
        IRateLimiter.OutboundQueuedTransfer memory qt = nttManagerChain1.getOutboundQueuedTransfer(0);
        assertEq(qt.amount.getAmount(), transferAmount.trim(decimals, decimals).getAmount());
        assertEq(qt.recipientChain, chainId2);
        assertEq(qt.recipient, toWormholeFormat(userB));
        assertEq(qt.txTimestamp, initialBlockTimestamp);

        // assert that the contract also locked funds from the user
        assertEq(token1.balanceOf(address(userA)), 0);
        assertEq(token1.balanceOf(address(nttManagerChain1)), transferAmount);


        // elapse some random time - 60 seconds
        uint256 durationElapsedTime = initialBlockTimestamp + 60;

        // now add a third transceiver
        nttManagerChain1.setTransceiver(address(wormholeTransceiver3Chain1));

        // verify that the third transceiver is added
        assertEq(nttManagerChain1.getTransceivers().length, 3);

        // remove second transceiver
        nttManagerChain1.removeTransceiver(address(wormholeTransceiver2Chain1));

          // verify that the second transceiver is removed
        assertEq(nttManagerChain1.getTransceivers().length, 2);

         // elapse rate limit duration
         durationElapsedTime = initialBlockTimestamp + nttManagerChain1.rateLimitDuration();

        vm.warp(durationElapsedTime);
```

```
        vm.expectRevert(stdError.indexOOBError); //index out of bounds - transceiver instructions array
        ↪  does not have a third element to access
        nttManagerChain1.completeOutboundQueuedTransfer(0);
    }

    // Encode an instruction for each of the relayers
  function encodeTransceiverInstructions(bool relayer_off) public view returns (bytes memory) {
        WormholeTransceiver.WormholeTransceiverInstruction memory instruction =
            IWormholeTransceiver.WormholeTransceiverInstruction(relayer_off);

        bytes memory encodedInstructionWormhole =
            wormholeTransceiverChain1.encodeWormholeTransceiverInstruction(instruction);

        TransceiverStructs.TransceiverInstruction memory TransceiverInstruction1 =
        TransceiverStructs.TransceiverInstruction({index: 0, payload: encodedInstructionWormhole});
        TransceiverStructs.TransceiverInstruction memory TransceiverInstruction2 =
        TransceiverStructs.TransceiverInstruction({index: 1, payload: encodedInstructionWormhole});

        TransceiverStructs.TransceiverInstruction[] memory TransceiverInstructions =
            new TransceiverStructs.TransceiverInstruction[](2);

        TransceiverInstructions[0] = TransceiverInstruction1;
        TransceiverInstructions[1] = TransceiverInstruction2;

        return TransceiverStructs.encodeTransceiverInstructions(TransceiverInstructions);
    }
}
```

Recommended Mitigation: Consider passing no instructions into the delivery price estimation when the Transceiver index does not exist.

7.2 Medium Risk

7.2.1 Silent overflow in `TrimmedAmount::shift` could result in rate limiter being bypassed

Description: Within `TrimmedAmount::trim`, there is an explicit check that ensures the scaled amount does not exceed the maximum `uint64`:

```
// NOTE: amt after trimming must fit into uint64 (that's the point of
// trimming, as Solana only supports uint64 for token amts)
if (amountScaled > type(uint64).max) {
    revert AmountTooLarge(amt);
}
```

However, no such check exists within `TrimmedAmount::shift` which means there is potential for silent overflow when casting to `uint64` here:

```
function shift(
    TrimmedAmount memory amount,
    uint8 toDecimals
) internal pure returns (TrimmedAmount memory) {
    uint8 actualToDecimals = minUint8(TRIMMED_DECIMALS, toDecimals);
    return TrimmedAmount(
        uint64(scale(amount.amount, amount.decimals, actualToDecimals)), actualToDecimals
    );
}
```

Impact: A silent overflow in `TrimmedAmount::shift` could result in the rate limiter being bypassed, considering its usage in `NttManager::_transferEntryPoint`. Given the high impact and reasonable likelihood of this issue occurring, it is classified a **MEDIUM** severity finding.

Recommended Mitigation: Explicitly check the scaled amount in `TrimmedAmount::shift` does not exceed the maximum `uint64`.

7.2.2 Disabled Transceivers cannot be re-enabled by calling `TransceiverRegistry::_setTransceiver` after 64 have been registered

Description: `TransceiverRegistry::_setTransceiver` handles the registering of Transceivers, but note that they cannot be re-registered as this has other downstream effects, so this function is also responsible for the re-enabling of previously registered but currently disabled Transceivers.

```
function _setTransceiver(address transceiver) internal returns (uint8 index) {
    /* snip */
    if (transceiver == address(0)) {
        revert InvalidTransceiverZeroAddress();
    }

    if (_numTransceivers.registered >= MAX_TRANSCEIVERS) {
        revert TooManyTransceivers();
    }

    if (transceiverInfos[transceiver].registered) {
        transceiverInfos[transceiver].enabled = true;
    } else {
    /* snip */
}
```

This function reverts if the passed transceiver address is `address(0)` or the number of registered transceivers is already at its defined maximum of 64. Assuming a total of 64 registered Transceivers, with some of these

Transceivers having been previously disabled, the placement of this latter validation will prevent a disabled Transceiver from being re-enabled since the subsequent block in which the storage indicating its enabled state is set to `true` is not reachable. Consequently, it will not be possible to re-enable any disabled transceivers after having registered the maximum number of Transceivers, meaning that this function will never be callable without redeployment.

Impact: Under normal circumstances, this maximum number of registered Transceivers should never be reached, especially since the underlying Transceivers are upgradeable. However, while unlikely based on operational assumptions, this undefined behavior could have a high impact, and so this is classified as a **MEDIUM** severity finding.

Recommended Mitigation: Move the placement of the maximum Transceivers validation to within the `else` block that is responsible for handling the registration of new Transceivers.

7.2.3 NTT Manager cannot be unpaused once paused

Description: `NttManagerState::pause` exposes pause functionality to be triggered by permissioned actors but has no corresponding unpause functionality. As such, once the NTT Manager is paused, it will not be possible to unpause without a contract upgrade.

```
function pause() public onlyOwnerOrPauser {
    _pause();
}
```

Impact: The inability to unpause the NTT Manager could result in significant disruption, requiring either a contract upgrade or complete redeployment to resolve this issue.

Recommended Mitigation:

```
+ function unpause() public onlyOwnerOrPauser {
+     _unpause();
+ }
```

7.2.4 Immutable gas limit within `WormholeTransceiver` can lead to execution failures on the target chain

Description: The Wormhole-specific Transceiver implementation uses an immutable `gasLimit` variable to calculate the Relayer delivery price. The underlying assumption here is that the gas consumed for transfers will always be static; however, this is not always the case, especially for L2 rollups such as Arbitrum, where gas is calculated as a function of the actual gas consumed on L2 and the L1 calldata cost that is effectively an L2 view of the L1 gas price. Please refer to the Arbitrum docs for more information on how gas is estimated.

In cases where L2 gas depends on the L1 gas price, extreme scenarios can occur where the delivery cost computed by the static gas limit is insufficient to execute a transfer on L2.

Impact: An immutable gas limit can give an extremely stale view of the L2 gas needed to execute a transfer. In extreme scenarios, such stale gas estimates can be insufficient to execute messages on a target chain. If such a scenario occurs, all pending messages with a stale gas estimate will risk being stuck on the target chain. While the gas limit can be changed via an upgrade, there are two issues with this approach:

1. Synchronizing a mass update across a large number of `NttManager` contracts might be difficult to execute.
2. Changing the gas limit has no impact on pending transfers that are already ready for execution on the target chain.

Recommended Mitigation: Consider making the gas limit mutable. If necessary, NTT Managers can keep track of L1 gas prices and change the gas limits accordingly.

7.2.5 Transceiver invariants and ownership synchronicity can be broken by unsafe Transceiver upgrades

Description: Transceivers are upgradeable contracts integral to the cross-chain message handling of NTT tokens. While `WormholeTransceiver` is a specific implementation of the `Transceiver` contract, NTT Managers can integrate with Transceivers of any custom implementation.

`Transceiver::_checkImmutables` is an internal virtual function that verifies that invariants are not violated during an upgrade. Two checks in this function are that a) the NTT Manager address remains the same and b) the underlying NTT token address remains the same.

However, the current logic allows integrators to bypass these checks by either:

1. Overriding the `_checkImmutables()` function without the above checks.
2. Calling `Implementation::_setMigratesImmutables` with a `true` input. This effectively bypasses the `_checkImmutables()` function validation during an upgrade.

Based on the understanding that Transceivers are deployed by integrators external to NTT Manager owners, regardless of the high trust assumptions associated with integrators, it is risky for NTT Managers to delegate power to Transceivers to silently upgrade a transceiver contract that can potentially violate the NTT Manager invariants.

One example of this involves the intended ownership model. Within `Transceiver::_initialize`, the owner of the Transceiver is set to the owner of the `NttManager` contract:

```
function _initialize() internal virtual override {
    // check if the owner is the deployer of this contract
    if (msg.sender != deployer) {
        revert UnexpectedDeployer(deployer, msg.sender);
    }

    __ReentrancyGuard_init();
    // owner of the transceiver is set to the owner of the nttManager
    __PausedOwnable_init(msg.sender, getNttManagerOwner());
}
```

However, the transferring of this ownership via `Transceiver::transferTransceiverOwnership` is only allowed by the NTT Manager itself:

```
/// @dev transfer the ownership of the transceiver to a new address
/// the nttManager should be able to update transceiver ownership.
function transferTransceiverOwnership(address newOwner) external onlyNttManager {
    _transferOwnership(newOwner);
}
```

When the owner of the NTT Manager is changed by calling `NttManagerState::transferOwnership`, the owner of all the Transceivers is changed with it:

```
/// @notice Transfer ownership of the Manager contract and all Endpoint contracts to a new owner.
function transferOwnership(address newOwner) public override onlyOwner {
    super.transferOwnership(newOwner);
    // loop through all the registered transceivers and set the new owner of each transceiver to the
    ↪  newOwner
    address[] storage _registeredTransceivers = _getRegisteredTransceiversStorage();
    _checkRegisteredTransceiversInvariants();

    for (uint256 i = 0; i < _registeredTransceivers.length; i++) {
        ITransceiver(_registeredTransceivers[i]).transferTransceiverOwnership(newOwner);
    }
}
```

This design is intended to ensure that the NTT Manager's owner is kept in sync across all transceivers, access-controlled to prevent unauthorized ownership changes, but transceiver ownership can still be transferred directly as the public `OwnableUpgradeable::transferOwnership` function has not been overridden. Even if Transceiver ownership changes, the Manager is permitted to change it again via the above function.

However, this behavior can be broken if the new owner of a Transceiver performs a contract upgrade without the immutables check. In this way, they can change the NTT Manager, preventing the correct manager from having permissions as expected. As a result, `NttManagerState::transferOwnership` will revert if any one Transceiver is out of sync with the others, and since it is not possible to remove an already registered transceiver, this function will cease to be useful. Instead, each Transceiver will be forced to be manually updated to the new owner unless the modified Transceiver is reset back to the previous owner so that this function can be called again.

Impact: While this issue may require the owner of a Transceiver to misbehave, a scenario where a Transceiver is silently upgraded with a new NTT Manager or NTT Manager token can be problematic for cross-chain transfers and so is prescient to note.

Proof of Concept: The below PoC calls the `_setMigratesImmutables()` function with the `true` boolean, effectively bypassing the `_checkImmutables()` invariant check. As a result, a subsequent call to `NttManagerState::transferOwnership` is demonstrated to revert. This test should be added to the contract in `Upgrades.t.sol` before running, and the revert in `MockWormholeTransceiverContract::transferOwnership` should be removed to reflect the true functionality.

```
function test_immutableUpgradePoC() public {
    // create the new mock ntt manager contract
    NttManager newImpl = new MockNttManagerContract(
        nttManagerChain1.token(), IManagerBase.Mode.BURNING, chainId1, 1 days, false
    );
    MockNttManagerContract newNttManager =
        MockNttManagerContract(address(new ERC1967Proxy(address(newImpl), "")));
    newNttManager.initialize();

    // transfer transceiver ownership
    wormholeTransceiverChain1.transferOwnership(makeAddr("new transceiver owner"));

    // create the new transceiver implementation, specifying the new ntt manager
    WormholeTransceiver wormholeTransceiverChain1Implementation = new
    ↪  MockWormholeTransceiverImmutableAllow(
        address(newNttManager),
        address(wormhole),
        address(relayer),
        address(0x0),
        FAST_CONSISTENCY_LEVEL,
        GAS_LIMIT
    );

    // perform the transceiver upgrade
    wormholeTransceiverChain1.upgrade(address(wormholeTransceiverChain1Implementation));

    // ntt manager ownership transfer should fail and revert
    vm.expectRevert(abi.encodeWithSelector(ITransceiver.CallerNotNttManager.selector, address(this)));
    nttManagerChain1.transferOwnership(makeAddr("new ntt manager owner"));
}
```

Recommended Mitigation: Consider making `Transceiver::_checkImmutables` and `Implementation::_setMigratesImmutables` private functions for Transceivers. If the `_checkImmutables()` function has to be overridden, consider exposing another function that is called inside `_checkImmutables` as follows:

```
function _checkImmutables() private view override {
    assert(this.nttManager() == nttManager);
    assert(this.nttManagerToken() == nttManagerToken);
```

```
    _checkAdditionalImmutables();
}

function _checkAdditionalImmutables() private view virtual override {}
```

7.2.6 Setting a peer `NttManager` contract for the same chain can cause loss of user funds

Description: The current implementation of `NttManager::setPeer` allows the owner to set the NTT Manager as a peer for the same chain ID as the current chain. If the NTT Manager owner accidentally (or otherwise) sets an arbitrary address as a peer `NttManager` address for the same chain, this configuration would allow a user to initiate a transfer with the same target chain as the source chain, but such transfers will not get executed on the target chain (which is same as source chain).

Impact: There is a potential loss of funds for the users. Even if the peer is subsequently removed, messages already sent from the source chain can never be executed. Any funds attached to those messages will be stuck in the `NttManager` contract.

Proof of Concept: The following test case shows that transactions fail on the destination chain.

```
function testTransferToOwnChain() external {
    uint16 localChainId = nttManager.chainId();
    DummyToken token = DummyToken(nttManager.token());
   uint8 decimals = token.decimals();
    nttManager.setPeer(localChainId, toWormholeFormat(address(0x999), decimals);

    address user_A = address(uint160(uint256(keccak256("user_A"))));
    address user_B = address(uint160(uint256(keccak256("user_B"))));

    token.mintDummy(address(user_A), 5 * 10 ** decimals);
    uint256 outboundLimit = 4 * 10 ** decimals;
    nttManager.setOutboundLimit(outboundLimit);

    // chk params before transfer
    IRateLimiter.RateLimitParams memory inboundLimitParams =
        nttManager.getInboundLimitParams(localChainId);
    IRateLimiter.RateLimitParams memory outboundLimitParams =
        nttManager.getOutboundLimitParams();

    assertEq(outboundLimitParams.limit.getAmount(), outboundLimit.trim(decimals, decimals).getAmount());
    assertEq(outboundLimitParams.currentCapacity.getAmount(), outboundLimit.trim(decimals,
    ↪  decimals).getAmount());
    assertEq(inboundLimitParams.limit.getAmount(), 0);
    assertEq(inboundLimitParams.currentCapacity.getAmount(), 0);

    vm.startPrank(user_A);
    uint256 transferAmount = 3 * 10 ** decimals;
    token.approve(address(nttManager), transferAmount);
    nttManager.transfer(transferAmount, localChainId, toWormholeFormat(user_B), false, new bytes(1));

    vm.stopPrank();


    // chk params after transfer

    // assert outbound rate limit has decreased
    outboundLimitParams = nttManager.getOutboundLimitParams();
    assertEq(
        outboundLimitParams.currentCapacity.getAmount(),
        (outboundLimit - transferAmount).trim(decimals, decimals).getAmount()
    );
```

```
    assertEq(outboundLimitParams.lastTxTimestamp, initialBlockTimestamp);

    // assert inbound rate limit for destination chain is still 0.
    // the backflow should not override the limit.
    inboundLimitParams =
        nttManager.getInboundLimitParams(localChainId);


    assertEq(inboundLimitParams.limit.getAmount(), 0);
    assertEq(inboundLimitParams.currentCapacity.getAmount(), 0);
}

function testSameChainEndToEndTransfer() public {
    vm.chainId(chainId1);

    // Setting up the transfer
    DummyToken token1 = DummyToken(nttManagerChain1.token());

    uint8 decimals = token1.decimals();
    uint256 sendingAmount = 5 * 10 ** decimals;
    token1.mintDummy(address(userA), 5 * 10 ** decimals);
    vm.startPrank(userA);
    token1.approve(address(nttManagerChain1), sendingAmount);

    vm.recordLogs();

    // Send token through standard means (not relayer)
    {

        uint256 userBalanceBefore = token1.balanceOf(address(userA));
        nttManagerChain1.transfer(sendingAmount, chainId1, bytes32(uint256(uint160(userB))));

        // Balance check on funds going in and out working as expected
        uint256 userBalanceAfter = token1.balanceOf(address(userA));

        require(
            userBalanceBefore - sendingAmount == userBalanceAfter,
            "User should have sent tokens"
        );
    }

    vm.stopPrank();

    // Get and sign the log to go down the other pipe. Thank you to whoever wrote this code in the past!
    Vm.Log[] memory entries = guardian.fetchWormholeMessageFromLog(vm.getRecordedLogs());
    bytes[] memory encodedVMs = new bytes[](entries.length);
    for (uint256 i = 0; i < encodedVMs.length; i++) {
        encodedVMs[i] = guardian.fetchSignedMessageFromLogs(entries[i], chainId1);
    }

    {
        uint256 supplyBefore = token1.totalSupply();

        vm.expectRevert(abi.encodeWithSelector(UnexpectedRecipientNttManagerAddress.selector,
        ↪  toWormholeFormat(address(nttManagerChain1)), toWormholeFormat(address(0x999))));
        wormholeTransceiverChain1.receiveMessage(encodedVMs[0]);
        uint256 supplyAfter = token1.totalSupply();

        // emit log_named_uint("sending amount on chain 1", sendingAmount);
        // emit log_named_uint("minting amount on chain 1", supplyAfter - supplyBefore);
        // // require(supplyAfter > sendingAmount + supplyBefore , "Supplies dont match");
        // // require(token1.balanceOf(userB) == sendingAmount, "User didn't receive tokens");
```

```
    }
}
```

Recommended Mitigation: Using cross-chain infrastructure to make transfers within the same chain makes little sense and can lead to loss of user funds if configured in this way. Consider preventing NTT Manager owners from setting peers for the same chain.

7.2.7 Lack of a gas refund in the current design can lead to the overcharging of users and misaligned relayer incentives that can choke message execution

Description: To understand gas handling, it is important to highlight a few key aspects of the current design:

1. On the target chain, Transceivers can either attest to a message or attest and execute a message. Transceivers up to the threshold attest to a message, while the Transceiver who will cause the threshold to be reached attests and executes a message.
2. Each transceiver quotes a gas estimate on the source chain. When quoting a price, no Transceiver knows if its peer on the target chain will simply attest to a message or both attest and execute a message. This means that every Transceiver quotes a gas estimate that assumes its peer will be executing a message.

Based on the two above facts, the following can be deduced:

1. If the threshold has not yet been reached, a sender is paying a delivery fee for every Transceiver, even ones that are attesting a message after it was already executed.
2. The sender is paying for a scenario where every Transceiver is responsible for executing a message on the target chain. In reality, only one transceiver will execute, and all others will attest.
3. If a relayer consistently calls a Transceiver before the threshold is reached, a relayer will earn more than is spent in terms of gas.
4. The above point incentivizes relayers to always be the ones to attest and not to execute. A clever relayer can simply query `messageAttestations` off-chain and skip a delivery if `messageAttestations == threshold - 1`, since a relayer spends less gas than what they charged on the source chain if they deliver before OR after a threshold is met.

Impact:

1. Users are overcharged on the source chain without recourse due to the lack of a refund mechanism.
2. Relayers can choke message execution by skipping execution of the message that meets the attestation threshold. Current economic incentives benefit relayers if they skip this specific message.

Recommended Mitigation: In the case of standard relayers, consider a mechanism to refund excess gas to the recipient address on the target chain. `DeliveryProvider:: quoteEvmDeliveryPrice` in the core Wormhole codebase returns a `targetChainRefundPerUnitGasUnused` parameter that is currently unused. Consider using this input to calculate the excess fee that can be refunded to the senders. Doing so will not only save costs for users but also remove any misaligned economic incentives for the relayers.

7.2.8 Access-controlled functions cannot be called when L2 sequencers are down

Description: Given that rollups such as Optimism and Arbitrum offer methods for forced transaction inclusion, it is important that the aliased sender address is also checked within access control modifiers when verifying the sender holds a permissioned role to allow the functions to which they are applied to be called even in the event of sequencer downtime. The most pertinent examples include:

- `PausableOwnable::_checkOwnerOrPauser`, which is called in the `onlyOwnerOrPauser` modifier, which itself is applied to `PausableOwnable::transferPauserCapability` and, more importantly, `NttManagerState::pause`.
- `OwnableUpgradeable::_checkOwner` which is called in the `onlyOwner` modifier, which itself is applied to `OwnableUpgradeable::transferOwnership`, `NttManagerState::upgrade`,

`NttManagerState::transferOwnership`, `NttManagerState::setTransceiver`, `NttManagerState::removeTransceiver`, `NttManagerState::setThreshold`, `NttManagerState::setPeer`, `NttManagerState::setOutboundLimit`, `NttManagerState::setInboundLimit`, and `Transceiver::upgrade`.

- The `onlyRelayer` modifier which is applied to `WormholeTransceiver::receiveWormholeMessages`.

Impact: Failure to consider the aliased sender address prevents the execution of admin or otherwise permissioned functionality on a chain where transactions are batched by a centralized L2 sequencer. Since this functionality could be time-sensitive, such as the urgent pausing of the protocol or the relaying of NTT messages, this issue has the potential to have a high impact with reasonable likelihood.

Proof of Concept: While potentially unlikely, a possible scenario could include:

1. An attacker identifies an exploit in the NTT Manager's protocol on the source chain and plans to bridge stolen funds to the Ethereum mainnet. Assume the source chain is an L2 rollup that batches transactions for publishing onto the L1 chain via a centralized sequencer.
2. The L2 sequencer goes down; however, transactions can still be executed via forced inclusion on the L1 chain. The Attacker could either have forced or waited for this to happen.
3. The Attacker exploits the protocol and initiates a native token transfer, kicking off the 24-hour rate limit duration. Assume the outbound rate limit is hit on the L2, but the inbound limit is not exceeded on the Ethereum mainnet.
4. Access-controlled functions are not callable since they do not check the aliased sender address, so admin transactions cannot be force-included from L1, e.g. pause the protocol.
5. The rate limit duration passes, and the sequencer is still down – the attacker completes the outbound transfer (assuming attestations are made) and relays the message on the Ethereum mainnet.

Recommended Mitigation: Validation of the sender address against permissioned owner/pauser/relayer roles should also consider the aliased equivalents to allow access-controlled functionality to be executed via forced inclusion. Another relevant precaution for the exploit case described above is to reduce the inbound rate limit of the affected chain to zero, which should work to mitigate this issue so long as the transaction can be successfully executed on the destination (i.e. it is not also an L2 rollup simultaneously experiencing sequencer downtime).

7.3 Low Risk

7.3.1 Inconsistent implementation of ERC-7201 Namespaced Storage locations

Description: Regarding the implementation of Namespaced Storage locations, the EIP-7201 formula should be followed to strictly conform to the EIP specification, as is correctly done by the modified OpenZeppelin libraries in `libraries/external`; elsewhere, usage is incorrect.

Additionally, there are instances where the location strings appear inconsistent, for example, `Pause.pauseRole`, which should likely instead be `Pauser.pauserRole`.

Impact: Namespaces are implemented to avoid collisions with other namespaces or the standard Solidity storage layout. The formula defined in EIP-7201 guarantees this property for arbitrary namespace IDs under the assumption of keccak256 collision resistance. It is, therefore, important to ensure that its implementation is consistent to fully benefit from these assurances.

Recommended Mitigation: Use the formula outlined in the EIP to update all other instances in `src/*`:

```
keccak256(abi.encode(uint256(keccak256("example.location")) - 1)) & ~bytes32(uint256(0xff));
```

7.3.2 Asymmetry in Transceiver pausing capability

Description: Pausing functionality is exposed via `Transceiver::_pauseTransceiver`; however, there is no corresponding function that exposes unpausing functionality:

```
/// @dev pause the transceiver.
function _pauseTransceiver() internal {
    _pause();
}
```

Impact: While not an immediate issue since the above function is not currently in use anywhere, this should be resolved to avoid cases where Transceivers could become permanently paused.

Recommended Mitigation:

```
+ /// @dev unpause the transceiver.
+ function _unpauseTransceiver() internal {
+     _unpause();
+ }
```

7.3.3 Incorrect Transceiver payload prefix definition

Description: The `WH_TRANSCEIVER_PAYLOAD_PREFIX` constant in `WormholeTransceiverState.sol` contains invalid ASCII bytes and, as such, does not match what is written in the inline developer documentation:

```
/// @dev Prefix for all TransceiverMessage payloads
///      This is 0x99'E''W''H'
/// @notice Magic string (constant value set by messaging provider) that idenfies the payload as an
↪  transceiver-emitted payload.
///         Note that this is not a security critical field. It's meant to be used by messaging
↪  providers to identify which messages are Transceiver-related.
bytes4 constant WH_TRANSCEIVER_PAYLOAD_PREFIX = 0x9945FF10;
```

The correct payload prefix is `0x99455748`, which is output when running the following command:

```
cast --from-utf8 "EWH"
```

Impact: While still a valid 4-byte hex prefix, used purely for identification purposes, an incorrect prefix could cause downstream confusion and result in otherwise valid Transceiver payloads being incorrectly prefixed.

Recommended Mitigation: Update the constant definition to use the correct prefix corresponding to the documented string:

```
+ bytes4 constant WH_TRANSCEIVER_PAYLOAD_PREFIX = 0x99455748;
```

7.3.4 `WormholeTransceiver` EVM chain IDs storage cannot be updated

In the unlikely but possible scenario in which a registered EVM-compatible chain diverges as the result of an upgrade, the existing implementation of `WormholeTransceiverState::setIsWormholeEvmChain` means that it will not be possible to update the corresponding storage. If, for whatever reason, a chain such as Optimism decides to move away from EVM (the opposite of what happened in reality, going from OVM to EVM), its chain ID will now correspond to a non-EVM chain. As such, this function should take a boolean argument, similar to the other functions defined below this one:

```
- function setIsWormholeEvmChain(uint16 chainId) external onlyOwner {
+ function setIsWormholeEvmChain(uint16 chainId, bool isEnabled) external onlyOwner {
    if (chainId == 0) {
        revert InvalidWormholeChainIdZero();
    }
-     _getWormholeEvmChainIdsStorage()[chainId] = TRUE;
+     _getWormholeEvmChainIdsStorage()[chainId] = toWord(isEnabled);

-     emit SetIsWormholeEvmChain(chainId);
+     emit SetIsWormholeEvmChain(chainId, isEnabled);
}
```

The `SetIsWormholeEvmChain` event will also need to be modified to take this additional boolean field.

7.3.5 Missing threshold invariant check when adding/removing transceivers

Description: `NttManagerState::_checkThresholdInvariants` is intended to check invariant properties related to the threshold storage and is called on initialization, migration, and access-controlled setting of the threshold storage directly. However, this function is not currently called when the `setTransceiver()` and removeTransceiver() functions execute, which also access and modify the relevant state.

Impact: A bug in the addition/removal of transceivers could go unnoticed since the inline invariant checks on the threshold storage are not being performed here despite being modified.

Recommended Mitigation: Ensure `NttManagerState::_checkThresholdInvariants` is called within `NttManagerState::setTransceiver` and `NttManagerState::removeTransceiver`.

7.4 Informational

7.4.1 Unchained initializers should be called in `PausableOwnable::__PausedOwnable_init` instead

While not an immediate issue in the current implementation, the direct use of initializer functions rather than their unchained equivalents should be avoided. These have been implemented and used in some places but not others, for example, `__PausedOwnable_init` which should be modified to avoid potential duplicate initialization in future.

```
function __PausedOwnable_init(address initialPauser, address owner) internal onlyInitializing {
    __Paused_init(initialPauser);
    __Ownable_init(owner);
}
```

7.4.2 Incorrect topics[0] documented in `INTTManagerEvents` and `IRateLimiterEvents`

Description: Inline NatSpec documentation incorrectly specifies topics[0] for the following events:

1. `INTTManagerEvents::TransceiverAdded` – Documented: `0xc6289e62021fd0421276d06677862d6b328d9764cdd4490ca5a`
   Correct: `0xf05962b5774c658e85ed80c91a75af9d66d2af2253dda480f90bce78aff5eda5`.
2. `INTTManagerEvents::TransceiverRemoved` – Documented: `0x638e631f34d9501a3ff0295873b29f50d0207b5400bf0e48b`
   Correct: `0x697a3853515b88013ad432f29f53d406debc9509ed6d9313dcfe115250fcd18f`.
3. `IRateLimiterEvents::OutboundTransferRateLimited` – Documented: `0x754d657d1363ee47d967b415652b739bfe96d57`
   Correct: `0xf33512b84e24a49905c26c6991942fc5a9652411769fc1e448f967cdb049f08a`.

7.4.3 NatSpec documentation inconsistent with function/event signatures

Inline NatSpec documentation does not reflect the actual signatures in multiple cases:

1. Missing definition for the `recipientNttManagerAddress` in the `ITransceiver::sendMessage` NatSpec:

```
/// @dev Send a message to another chain.
/// @param recipientChain The Wormhole chain ID of the recipient.
/// @param instruction An additional Instruction provided by the Transceiver to be
/// executed on the recipient chain.
/// @param nttManagerMessage A message to be sent to the nttManager on the recipient chain.
function sendMessage(
    uint16 recipientChain, //@note wormhole chain id
    TransceiverStructs.TransceiverInstruction memory instruction, //@note same as above
    bytes memory nttManagerMessage,
    bytes32 recipientNttManagerAddress
) external payable;
```

2. Missing definition for `sequence` in the `IRateLimiterEvents::OutboundTransferRateLimited` NatSpec:

```
/// @notice Emitted when an outbound transfer is rate limited.
/// @dev Topic0
///      0x754d657d1363ee47d967b415652b739bfe96d5729ccf2f26625dcdbc147db68b.
/// @param sender The initial sender of the transfer.
/// @param amount The amount to be transferred.
/// @param currentCapacity The capacity left for transfers within the 24-hour window.
event OutboundTransferRateLimited(
    address indexed sender, uint64 sequence, uint256 amount, uint256 currentCapacity
);
```

7.4.4 Unused `PausableUpgradeable::CannotRenounceWhilePaused` error should be removed

`PausableUpgradeable::CannotRenounceWhilePaused` is a custom error defined as follows:

```
/**
 * @dev Cannot renounce the pauser capability when the contract is in the `PAUSED` state
 */
error CannotRenounceWhilePaused(address account);
```

The above error and inline comments imply that the pauser capability cannot be transferred when a contract is in a `PAUSED` state. However, no such check is performed in `PausableOwnable::transferPauserCapability`:

```
/**
 * @dev Transfers the ability to pause to a new account (`newPauser`).
 */
function transferPauserCapability(address newPauser) public virtual onlyOwnerOrPauser {
    PauserStorage storage $ = _getPauserStorage();
    address oldPauser = $._pauser;
    $._pauser = newPauser;
    emit PauserTransferred(oldPauser, newPauser);
}
```

Given that it is understood this is not an error of omission, where stated functionality is not implemented in the function, but rather an unused custom error that is not intended to be used, it is recommended that the definition be removed.

7.4.5 Multiple `staticcall` success booleans are not checked

The success boolean returned when performing a `staticcall` on the given ERC20 token is not currently checked for any of the instances. It is understood this is a trust assumption in that legitimate implementations should always have `IERC20::decimals` implemented and should not revert when calling `IERC20::balanceOf`; however, it is recommended to check this returned value so as to guarantee the impossibility of erroneously using the revert data when expecting to decode something else.

```
function _getTokenBalanceOf(
    address tokenAddr,
    address accountAddr
) internal view returns (uint256) {
    (, bytes memory queriedBalance) =
        tokenAddr.staticcall(abi.encodeWithSelector(IERC20.balanceOf.selector, accountAddr));
    return abi.decode(queriedBalance, (uint256));
}
```

7.4.6 Canonical NTT chain ID should be fetched directly from Wormhole or mapped accordingly

Currently, the immutable `chainId` variable is assigned the value passed to the `NttManager` constructor by the deployer:

```
constructor(
    address _token,
    Mode _mode,
    uint16 _chainId,
    uint64 _rateLimitDuration
) NttManagerState(_token, _mode, _chainId, _rateLimitDuration) {}
```

However, no validation is performed to ensure that the chain identifier provided is as expected. Since this is intended to be the Wormhole chain ID, it should be validated against the value returned by the Wormhole contract. Otherwise, if the chain identifier does not match the Wormhole chain identifier, given the understanding that the ID is not necessarily required to be the Wormhole chain ID, then this state should be included within the `WormholeTransceiver` payload published to Wormhole since it will differ from the emitter chain included in the VAA.

If it is decided to make the Wormhole chain ID the canonical chain ID, different Transceiver implementations should map to the corresponding chain ID representations within their logic. As such, the Wormhole address should be passed to the constructor of the NTT Manager, querying the chain ID directly from the Wormhole Core Bridge instead of taking an arbitrary value in the constructor as described above.

7.5 Gas Optimization

7.5.1 Unnecessary type casts of proxied storage slots from `bytes32` to `uint256`

There are multiple instances where proxied storage slots are cast from type `bytes32` to `uint256`; however, this is not necessary as the EVM uses 32-byte words, meaning that these types are interchangeable when considering the subsequent inline assembly assignment. One example can be found in `Governance::_getConsumedGovernanceActionsStorage`:

```
function _getConsumedGovernanceActionsStorage()
    private
    pure
    returns (mapping(bytes32 => bool) storage $)
{
    uint256 slot = uint256(CONSUMED_GOVERNANCE_ACTIONS_SLOT);
    assembly ("memory-safe") {
        $.slot := slot
    }
}
```

Consider removing this type cast to save gas.

7.5.2 Unnecessary stack variable in `Governance::encodeGeneralPurposeGovernanceMessage`

`Governance::encodeGeneralPurposeGovernanceMessage` currently assigns the stack variable `callDataLength`; however, this is not necessary as the (already checked) downcast can be performed directly within the subsequent packed encoding.

```
function encodeGeneralPurposeGovernanceMessage(GeneralPurposeGovernanceMessage memory m)
    public
    pure
    returns (bytes memory encoded)
{
    if (m.callData.length > type(uint16).max) {
        revert PayloadTooLong(m.callData.length);
    }
    uint16 callDataLength = uint16(m.callData.length);
    return abi.encodePacked(
        MODULE,
        m.action,
        m.chain,
        m.governanceContract,
        m.governedContract,
        callDataLength,
        m.callData
    );
}
```

Consider removing this stack variable to save gas.

7.5.3 Optimize the `TrimmedAmount` struct as a bit-packed `uint72` user-defined type

The existing `TrimmedAmount` abstraction has a runtime gas overhead due to allocation of the struct:

```
struct TrimmedAmount {
    uint64 amount;
    uint8 decimals;
```

```
}
```

This could be mitigated by instead implementing a user-defined type that is a bit-packed `uint72` representation of a token amount and its decimals. The use of a `uint72` type ensures that the existing width of 9 bytes is maintained, which allows for tight packing elsewhere in storage, such as in the `RateLimitParams` struct:

```
struct RateLimitParams {
        TrimmedAmount limit;
        TrimmedAmount currentCapacity;
        uint64 lastTxTimestamp;
    }
```

therefore keeping this definition contained within a single word.

Despite the non-trivial refactoring effort required, a user-defined value type is advantageous here as it would allow for the trimmed amounts to be stack-allocated and arithmetic operators to be overloaded.

7.5.4 Optimize invariant check iterations in `TransceiverRegistry::_checkTransceiversInvariants`

The first loop over all enabled transceivers in `TransceiverRegistry::_checkTransceiversInvariants` could instead be included in the top level of the nested loop directly below to save multiple iterations and hence gas.

```
function _checkTransceiversInvariants() internal view {
    _NumTransceivers storage _numTransceivers = _getNumTransceiversStorage();
    address[] storage _enabledTransceivers = _getEnabledTransceiversStorage();

    uint256 numTransceiversEnabled = _numTransceivers.enabled;
    assert(numTransceiversEnabled == _enabledTransceivers.length);

-   for (uint256 i = 0; i < numTransceiversEnabled; i++) {
-       _checkTransceiverInvariants(_enabledTransceivers[i]);
-   }
-
    // invariant: each transceiver is only enabled once
    for (uint256 i = 0; i < numTransceiversEnabled; i++) {
+       _checkTransceiverInvariants(_enabledTransceivers[i]);
        for (uint256 j = i + 1; j < numTransceiversEnabled; j++) {
            assert(_enabledTransceivers[i] != _enabledTransceivers[j]);
        }
    }

    // invariant: numRegisteredTransceivers <= MAX_TRANSCEIVERS
    assert(_numTransceivers.registered <= MAX_TRANSCEIVERS);
}
```