Wormhole Monad NTT Audit Report

Prepared by Cyfrin

Version 2.1

Lead Auditors

Kage

Al-qaqa

November 20, 2025

Contents

1 About Cyfrin **2**

2 Disclaimer **2**

3 Risk Classification **2**

4 Protocol Summary **2**

5 Audit Scope **3**

6 Executive Summary **3**

7 Findings **5**

7.1 High Risk . . . 5

7.1.1 Denial of service in `MultiTokenNtt` if any transceiver has non-zero fees . . . 5

7.2 Medium Risk . . . 6

7.2.1 Missing fork protection implementation in `GmpManager` . . . 6

7.2.2 Tokens can be trapped when recipient callback fails . . . 6

7.2.3 Potential digest collision on inbound queued transfers can lead to loss of funds . . . 7

7.3 Low Risk . . . 8

7.3.1 `MultiTokenNtt::overrideLocalAsset` allows overriding native token representations . . . 8

7.3.2 `MultiTokenNtt::overrideLocalAsset` allows mapping single local token to multiple foreign assets, potentially corrupting internal accounting and rate limiting . . . 8

7.3.3 Incorrect error when burning fee token . . . 9

7.4 Informational . . . 10

7.4.1 ERC20 tokens not supporting decimals can't be used in NTT . . . 10

7.4.2 Incorrect invariant documentation in TransceiverRegistry . . . 10

7.4.3 Unused error types in `MultiTokenNtt.sol` . . . 10

7.4.4 Incorrect inline comments for `TransceiverAdded` event . . . 11

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

Wormhole's Native Token Transfer (NTT) framework is a cross-chain messaging protocol designed to enable secure and efficient token transfers across heterogeneous blockchain networks. The Monad integration extends NTT's capabilities to support Monad's high-performance EVM-compatible blockchain, providing a bridge for multi-token transfers.

MultiTokenNtt: The core contract managing multi-token bridging operations, including:

- Token registration and lifecycle management
- Cross-chain transfer initiation and receipt
- Rate limiting and security controls
- Fee collection and refund mechanisms
- Support for both native and ERC20 token transfers

GmpManager: Central orchestrator for cross-chain messaging that:

- Manages transceiver integrations for message delivery
- Handles fee aggregation and distribution across multiple transceivers
- Coordinates message sequencing and delivery confirmation
- Enforces security validation rules

Transceivers: Modular message transport layer supporting:

- Multiple concurrent transceivers with different security models
- Configurable threshold requirements for message attestation
- Dynamic fee calculation and routing
- Pluggable architecture for adding new transport mechanisms

Registry Components: Infrastructure for managing protocol state:

- *TransceiverRegistry*: Maintains active transceivers and their configurations
- *AssetRegistry*: Maps local tokens to their cross-chain representations
- Rate limiter integration for transfer volume controls

5 Audit Scope

The Wormhole Monad NTT audit encompasses the complete cross-chain token transfer infrastructure, focusing on the multi-token bridging implementation and its integration with Wormhole's General Message Passing (GMP) system. The protocol employs a modular transceiver architecture to facilitate secure message passing and token transfers across heterogeneous blockchain networks.

The audit concentrated on identifying security vulnerabilities in the token transfer mechanisms, message validation processes, fee calculation logic, and the overall integrity of cross-chain operations. Special attention was given to the multi-token management system, transceiver coordination, rate limiting controls, and the refund mechanisms for excess fees.

All files in the `evm/src` folder, specific to the EVM specific implementation, were part of the scope.

6 Executive Summary

Over the course of 15 days, the Cyfrin team conducted an audit on the Wormhole Monad NTT smart contracts provided by Wormhole Foundation. In this period, a total of 11 issues were found.

The Wormhole Monad Native Token Transfer (NTT) framework implements a sophisticated cross-chain token bridging system that enables seamless asset transfers between Monad and other blockchains in the Wormhole network.

The security audit examined the core bridging mechanics, multi-transceiver coordination, fee management systems, token registration workflows, and the protocol's resilience against common cross-chain attack vectors. The review encompassed all critical components including the MultiTokenNtt contract, GmpManager, transceiver implementations, and their associated security controls.

The audit revealed a well-architected codebase with robust security considerations. During the course of the audit, several issues were identified, the most significant of which is a denial of service vulnerability in the refund calculation mechanism.

The Wormhole Monad NTT protocol demonstrates strong engineering practices including:

- Modular architecture enabling flexible transceiver configurations
- Comprehensive rate limiting and security controls
- Well-structured fee management with refund capabilities
- Clear separation between message passing and token handling logic
- Robust validation of cross-chain messages with multi-transceiver attestation

Summary

Project Name	Wormhole Monad NTT
Repository	monad-ntt
Commit	4e1acc085f7b...
Repository 2	native-token-transfers
Commit	7b29f8f03f6f...
Audit Timeline	Sep 8th - Sep 26th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	3
Low Risk	3
Informational	4
Gas Optimizations	0
Total Issues	11

Summary of Findings

[H-1] Denial of service in `MultiTokenNtt` if any transceiver has non-zero fees	Resolved
[M-1] Missing fork protection implementation in `GmpManager`	Acknowledged
[M-2] Tokens can be trapped when recipient callback fails	Resolved
[M-3] Potential digest collision on inbound queued transfers can lead to loss of funds	Resolved
[L-1] `MultiTokenNtt::overrideLocalAsset` allows overriding native token representations	Resolved
[L-2] `MultiTokenNtt::overrideLocalAsset` allows mapping single local token to multiple foreign assets, potentially corrupting internal accounting and rate limiting	Resolved
[L-3] Incorrect error when burning fee token	Acknowledged
[I-1] ERC20 tokens not supporting decimals can't be used in NTT	Acknowledged
[I-2] Incorrect invariant documentation in TransceiverRegistry	Acknowledged
[I-3] Unused error types in `MultiTokenNtt.sol`	Acknowledged
[I-4] Incorrect inline comments for `TransceiverAdded` event	Acknowledged

7 Findings

7.1 High Risk

7.1.1 Denial of service in `MultiTokenNtt` if any transceiver has non-zero fees

Description: When users bridge tokens through MultiTokenNtt, the ETH payment follows this path:

- User sends ETH to `MultiTokenNtt`
- `MultiTokenNtt` forwards ETH to `GmpManager`
- `GmpManager` uses some ETH for transceiver fees, returns excess to `MultiTokenNtt`
- `MultiTokenNtt` should refund remaining ETH to user

The refund calculation in `_sendMessageWithSequence()` is fundamentally flawed:

```
    function _sendMessageWithSequence( ... ) internal returns (uint64 messageSequence) {
        ...
        // _sendMessage invokes the GmpManager contract which takes the payment
        // for sending the message (including paying the transceivers).
        // It then refunds any excess back to this contract, which we refund to
        // the sender of this transaction.
        uint256 balanceBefore = address(this).balance;
        messageSequence = _sendMessage( ... );
>>      uint256 refundAmount = address(this).balance - balanceBefore;
        _refundToSender(refundAmount);
    }
// ----------------
    function _sendMessage( ... ) internal returns (uint64 sequence) {
>>      return gmpManager.sendMessage{value: msgValue}(
            targetChain, callee, refundAddress, reservedSequence, data, transceiverInstructions
        );
    }
```

The issue is that `balanceBefore` includes the user's payment that was forwarded out, making it impossible for the final balance to ever be higher unless zero fees are charged.

Impact: Complete denial of service for all token transfers whenever any transceiver charges non-zero fees.

Proof of Concept: Change the `_quoteDeliveryPrice` in `GenericDummyTransceiver`

```
function _quoteDeliveryPrice(
        uint16, /* recipientChain */
        TransceiverStructs.TransceiverInstruction memory /* transceiverInstruction */
    ) internal pure override returns (uint256) {
        return 100;
    }
```

Now add the modified `testTransferNative` test to `MultiTokenNtt.t.sol` to get the underflow error:

```
    function testTransferNative2() public {
    uint256 amount = 100 * 10 ** 18;
    address sender = chain1.addr(user1);
    address recipient = chain2.addr(user1);

    MockERC20 chain1Token = _deployAndMintToken("Test Token", "TEST", sender, 10 * amount);

    vm.startPrank(sender);
    chain1Token.approve(address(chain1.ntt()), amount);

    uint256 deliveryFee = 100;
    vm.deal(sender, deliveryFee);
```

```
    (uint64 sequence,, TransceiverStructs.TransceiverMessage memory transceiverMessage) =
        _executeTransfer(chain1, chain2, recipient, address(chain1Token), amount, deliveryFee);

    // Rest of the test remains the same...
    vm.stopPrank();
}
```

Recommended Mitigation: Consider adding `msgValue` when calculating `refundAmount`

```
diff --git a/1-monad-ntt/evm/src/MultiTokenNtt/MultiTokenNtt.sol
↪  b/1-monad-ntt/evm/src/MultiTokenNtt/MultiTokenNtt.sol
index 512e4d8..d260dfe 100644
--- a/1-monad-ntt/evm/src/MultiTokenNtt/MultiTokenNtt.sol
+++ b/1-monad-ntt/evm/src/MultiTokenNtt/MultiTokenNtt.sol
@@ -723,7 +723,7 @@ contract MultiTokenNtt is
             message,
             transceiverInstructions
         );
-        uint256 refundAmount = address(this).balance - balanceBefore;
+        uint256 refundAmount = address(this).balance + msgValue - balanceBefore;
         _refundToSender(refundAmount);
     }
```

Wormhole: Fixed in commit 933a281

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 Missing fork protection implementation in `GmpManager`

Description: While `GmpManager` inherit the `evmChainId` immutable variable from `ManagerBase`, it fails to implement or utilize the fork protection mechanism that `NttManager` employs. The `checkFork` function and its usage is only present in `NttManager`, creating an inconsistency in security measures across contracts that handle critical cross-chain operations.

Impact: Without fork protection:

- User can initiate transfer of TokenA on Fork 1
- Same user can initiate transfer of TokenA on Fork 2
- Both transfers execute on target chain
- Tokens are effectively double-spent across chains

Proof of Concept: Add the following to `MultiTokenNtt.sol`

```
function test_doubleSpendingOnFork() public {
    // Setup
    Token originalToken = new Token();
    originalToken.initialize("Test Token", "TEST", 18);
    uint256 amount = 1 ether;
    address sender = chain1.addr(user1);
    address recipient = chain2.addr(user2);

    originalToken.setMinter(address(this));
    originalToken.mint(sender, amount * 2);

    // Pre-deploy wrapped token on chain2 to avoid deployment issues
    Token wrappedToken = new Token();
    wrappedToken.initialize("Test Token", "TEST", 18);
    wrappedToken.setMinter(address(chain2.ntt()));

    TokenId memory tokenId = TokenId({
        chainId: chain1.chainId(),
        tokenAddress: toWormholeFormat(address(originalToken))
    });

    chain2.ntt().overrideLocalAsset(tokenId, address(wrappedToken));

    vm.startPrank(sender);
    originalToken.approve(address(chain1.ntt()), 100 ether);

    // Transfer 1
    (, bytes memory payload1,) =
        _executeTransfer(chain1, chain2, recipient, address(originalToken), amount, 0);

    // Transfer 2: Simulating fork
    // In a real fork:
    // 1. The chain splits into two versions
    // 2. User has same balance on both forks
    // 3. User can execute same transfer on both forks
    // 4. Both messages get relayed to destination chain

    // For testing purposes, we simulate this by:
    // - simply changing chainId
    // - Executing another transfer (simulating the fork's duplicate state)
    // - This represents what would happen on the forked chain
    vm.chainId(block.chainid + 10); // Simulate a fork by changing chainId
    (, bytes memory payload2,) =
        _executeTransfer(chain1, chain2, recipient, address(originalToken), amount, 0);
```

```
    vm.stopPrank();

    // Process message 1 on chain2
    _processMessage(chain2, payload1);
    assertEq(
        wrappedToken.balanceOf(recipient),
        amount,
        "Amount transferred should be correct after first transfer"
    );

    // Process message 2 on chain2
    _processMessage(chain2, payload2);

    // recipient received 2x the amount
    assertEq(
        wrappedToken.balanceOf(recipient),
        amount * 2,
        "Double spending on recipient chain"
    );
}
```

Recommended Mitigation: Consider adding fork protection to `MultiTokenNtt` and `GmpManager`

Wormhole: Acknowledged.

Cyfrin: Acknowledged.

7.2.2 Tokens can be trapped when recipient callback fails

Description: `MultiTokenNtt` implements a callback pattern using `INttTokenReceiver.onNttTokenReceived()` when transfers include an additionalPayload. This design allows tokens to be trapped if the recipient contract's callback fails for any reason.

Consider following flow:

- User initiates transfer with payload, tokens are burned/locked
- Cross-chain message relayed to destination chain
- Contract attempts to mint/unlock tokens to recipient
- Contract calls `INttTokenReceiver(recipient).onNttTokenReceived()`
- If callback reverts then entire transaction reverts

Tokens never minted on destination, but already burned on source. It is worthwhile to note that this issue does NOT exist in the `NttManager`, which only allows token transfers

```
//NttManager::_mintOrUnlockToRecipient

// Transfer tokens, then make external call that can fail
_mintOrUnlock(recipient, amount);
      try INttTokenReceiver(recipient).onNttTokenReceived(
                token,
                fromWormholeFormat(sourceAddress),
                untrimmedAmount,
                additionalPayload,
                sourceChainId,
                sourceAddress
            ) {
                // Callback succeeded
            } catch {
                revert NttTokenReceiverCallFailed(recipient); //@audit entire txn reverts if payload
                ↪  cannot be executed
```

```
        }
```

Impact: There are several scenarios that can trigger this issue:

- Silent upgrades - recipient contract upgraded removing `INttTokenReceiver` interface or modifying the target function
- Contracts that implement pausability
- Gas limit issues - recipient might implement gas heavy logic in callback
- Malicious contracts that selectively revert
- Bug in recipient contracts that causes callback reverts

In all such instances, sender has already burnt or locked their funds on the source chain but cannot claim those funds on the target chain.

Proof of Concept: Add the following to `MultiTokenNtt.t.sol`:

```
contract AlwaysRevertReceiver is INttTokenReceiver {
    address public multiTokenNttContract;

    constructor(address _multiTokenNttContract) {
        multiTokenNttContract = _multiTokenNttContract;
    }

    function onNttTokenReceived(
        address,
        address,
        uint256,
        bytes calldata,
        uint16,
        bytes32
    ) external view override {

        revert("Always revert");
    }
}
```

Then add the test to `MultiTokenNttTest.t.sol`:

```
function test_callbackFailure_tokensStuckForever() public {
        // Setup: Create a receiver that will revert
        AlwaysRevertReceiver revertingReceiver = new AlwaysRevertReceiver(address(chain1.ntt()));
        vm.deal(user1, 100 ether);
        // Setup token
        MockERC20 token = new MockERC20("Test Token", "TEST", 18);
        uint256 amount = 100e18;
        token.mint(user1, amount);

        // Record initial balances
        uint256 user1BalanceBefore = token.balanceOf(user1);
        uint256 receiverBalanceBefore = token.balanceOf(address(revertingReceiver));

        // User attempts to transfer with payload to the reverting receiver
        vm.startPrank(user1);
        token.approve(address(chain1.ntt()), type(uint256).max);
        MultiTokenNtt.TransferArgs memory args = MultiTokenNtt.TransferArgs({
            token: address(token),
            amount: amount,
            recipientChain: OTHER_CHAIN_ID,
            recipient: toWormholeFormat(address(revertingReceiver)),
            refundAddress: toWormholeFormat(user1),
            shouldQueue: false,
```

```
            transceiverInstructions: "",
            additionalPayload: abi.encode("trigger callback") // This will cause the callback
        });

        // Transfer succeeds on source chain - tokens are burned/locked
        uint64 sequence = chain1.ntt().transfer{value: 1 ether}(args);
        vm.stopPrank();

        // Verify tokens were taken from user on source chain
        assertEq(token.balanceOf(user1), 0, "Tokens should be taken from user");

         // Get the sent message from chain1's transceiver
        GenericDummyTransceiver chain1Transceiver =
        GenericDummyTransceiver(chain1.gmpManager().getTransceivers()[0]);

           // Get the sent message
        bytes memory sentMessage = chain1Transceiver.messages(0);

         address chain2Transceiver = chain2.gmpManager().getTransceivers()[0];

        //Attempt to execute on destination chain - THIS WILL REVERT
        vm.expectRevert(
                abi.encodeWithSelector(
                    NttTokenReceiverCallFailed.selector,
                    address(revertingReceiver)
                )
            );

        vm.startPrank(relayer);
        DummyTransceiver(chain2Transceiver).receiveMessage(sentMessage);
        vm.stopPrank();

}
```

Recommended Mitigation: Consider implementing an `EXECUTION_DELAY` window - if a transfer cannot be executed even after `EXECUTION_DELAY` window expires, consider making callbacks non-reverting.

Alternatively, consider documenting risks of using callbacks clearly as part of the in-line comments and technical documentation.

Wormhole: Fixed in 644aee9.

Cyfrin: Verified.

7.2.3 Potential digest collision on inbound queued transfers can lead to loss of funds

Description: In the event a transfer execution on the target chain triggers an inbound rate limit, the transfer parameters are hashed to create a `digest`. This digest is then used when completing inbound transfers after rate limit duration expires.

In the edge case when the token, sender, recipient, transfer amount and payload are the same for two different transfers initiated from two different chains, and the token triggers inbound rate limits, there is a possibility of digest hash collision.

MultiTokenNtt.sol#L268-L269

```
    function _executeMsg( ... ) internal whenNotPaused nonReentrant {
>>      bytes32 digest =
            keccak256(NativeTokenTransferCodec.encodeNativeTokenTransfer(nativeTokenTransfer));

        ...
    }
```

`NativeTokenTransfer` digest creation does not include the source chainId (emitter chainId). This means the digest retrieval can be the same for different NTTs from different source chains. As a result, the second transfer added to the inbound queue can overwrite the existing transfer causing a token loss.

MultiTokenRateLimiter.sol#L218-L225

```
    function _enqueueInboundTransfer(bytes32 digest, uint16 sourceChainId) internal {
        _getInboundQueueStorage()[digest] = InboundQueuedTransfer({
            txTimestamp: uint64(block.timestamp),
            sourceChainId: sourceChainId
        });

        emit InboundTransferQueued(digest);
    }
```

Impact: In the event where the token, sender, recipient, transfer amount and payload combination is identical, queued inbound transfers can be overwritten leading to a loss of tokens.

Proof of Concept: Add the following test to `MultiTokenNtt.t.sol`:

```
function testDigestCollision() public {

        uint16 CHAIN_3_ID = 0x99;
        TestDeployment chain3 = new TestDeployment(CHAIN_3_ID);

        {
            // Configure chain3 , transceiver and ntt peers
            chain3.gmpManager().setPeer(CHAIN_ID, toWormholeFormat(address(chain1.gmpManager())));
            chain3.gmpManager().setPeer(OTHER_CHAIN_ID, toWormholeFormat(address(chain2.gmpManager())));
            chain1.gmpManager().setPeer(CHAIN_3_ID, toWormholeFormat(address(chain3.gmpManager())));
            chain2.gmpManager().setPeer(CHAIN_3_ID, toWormholeFormat(address(chain3.gmpManager())));
            chain3.addTransceiver();

            address chain1Transceiver = chain1.gmpManager().getTransceivers()[0];
            address chain3Transceiver = chain3.gmpManager().getTransceivers()[0];

            chain1.gmpManager().setSendTransceiverForChain(CHAIN_3_ID, chain1Transceiver);
            chain1.gmpManager().setReceiveTransceiverForChain(CHAIN_3_ID, chain1Transceiver);
            chain3.gmpManager().setSendTransceiverForChain(CHAIN_ID, chain3Transceiver);
            chain3.gmpManager().setReceiveTransceiverForChain(CHAIN_ID, chain3Transceiver);

            chain3.ntt().setPeer(CHAIN_ID, toWormholeFormat(address(chain1.ntt())));
            chain1.ntt().setPeer(CHAIN_3_ID, toWormholeFormat(address(chain3.ntt())));
        }


        address fixedSender = address(0x1111111111111111111111111111111111111111);
        address fixedRecipient = address(0x2222222222222222222222222222222222222222);
        //Deploy chain1 token ===
        MockERC20 hubToken = _deployAndMintToken("Hub Token", "HUB", fixedSender, 1000 * 10 ** 18);


        // Bridge token from hub to chain2 and chain3
        // This creates wrapped versions on both spoke chains

        {
            // Bridge to chain2
            vm.startPrank(fixedSender);
            hubToken.approve(address(chain1.ntt()), 1000 * 10 ** 18);
            (, bytes memory payloadToChain2,) =
                _executeTransfer(chain1, chain2, fixedSender, address(hubToken), 100 * 10 ** 18, 0);
            vm.stopPrank();
            _processMessage(chain2, payloadToChain2);
        }
```

```
// Bridge to chain3
{
    vm.startPrank(fixedSender);
    (, bytes memory payloadToChain3,) =
        _executeTransfer(chain1, chain3, fixedSender, address(hubToken), 100 * 10 ** 18, 0);
    vm.stopPrank();
    _processMessage(chain3, payloadToChain3);
}

// Set limits that will be exceeded by our test transfers
{
    TokenId memory tokenId = TokenId({
    chainId: chain1.chainId(),
    tokenAddress: toWormholeFormat(address(hubToken))
    });

    // Set low inbound limits from both chain2 and chain3
    vm.prank(chain1.ntt().owner());
    chain1.ntt().setInboundLimit(tokenId, 5 * 10 ** 18, OTHER_CHAIN_ID); // chain2 limit: 5
    ↪  tokens
    vm.prank(chain1.ntt().owner());
    chain1.ntt().setInboundLimit(tokenId, 5 * 10 ** 18, CHAIN_3_ID);     // chain3 limit: 5
    ↪  tokens
}

// Both will exceed rate limits and should be queued
uint256 transferAmount = 75 * 10 ** 18; // Exceeds both limits
bytes32 digest;

// get wrapped tokens
Token wrappedOnChain2 = _getWrappedToken(chain2, address(hubToken), chain1.chainId());
Token wrappedOnChain3 = _getWrappedToken(chain3, address(hubToken), chain1.chainId());


// Execute transfer from chain2 (should be rate limited and queued)
{
    vm.warp(1000);
    vm.startPrank(fixedSender);
    wrappedOnChain2.approve(address(chain2.ntt()), transferAmount);
    (, bytes memory msgFromChain2,) = _executeTransfer(
        chain2,
        chain1,
        fixedRecipient,
        address(wrappedOnChain2),
        transferAmount,
        0
    );
    vm.stopPrank();
    // Process on chain1 - should be queued due to rate limit
    _processMessage(chain1, msgFromChain2);
    // Calculate digest from the transfer
    (, TransceiverStructs.TransceiverMessage memory parsed) = _extractTransferMessage(chain2);
    digest = _calculateDigestFromParsedMessage(parsed);
}


// Check that transfer from chain2 was queued
{
    IMultiTokenRateLimiter.InboundQueuedTransfer memory queued1 =
        chain1.ntt().getInboundQueuedTransfer(digest);
    assertEq(queued1.sourceChainId, OTHER_CHAIN_ID, "Should be queued from chain2");
```

```
            assertEq(queued1.txTimestamp, 1000, "Should have timestamp 1000");
        }

        // execute transfer on chain 3 - move timestamp to 2000
        {
            vm.warp(2000);
            vm.startPrank(fixedSender);
            wrappedOnChain3.approve(address(chain3.ntt()), transferAmount);
            (, bytes memory msgFromChain3,) = _executeTransfer(
                chain3,
                chain1,
                fixedRecipient,
                address(wrappedOnChain3),
                transferAmount,
                0
            );
            vm.stopPrank();

            _processMessage(chain1, msgFromChain3);
        }


        IMultiTokenRateLimiter.InboundQueuedTransfer memory queued2 =
            chain1.ntt().getInboundQueuedTransfer(digest);

        assertEq(queued2.sourceChainId, CHAIN_3_ID, "Chain2 entry was overwritten by chain3!");
        assertEq(queued2.txTimestamp, 2000, "Timestamp was overwritten!");
    }

    function _calculateDigestFromParsedMessage(
        TransceiverStructs.TransceiverMessage memory parsed
    ) internal pure returns (bytes32) {
        TransceiverStructs.NttManagerMessage memory nttMsg =
            TransceiverStructs.parseNttManagerMessage(parsed.nttManagerPayload);
        GmpStructs.GenericMessage memory genMsg =
            GmpStructs.parseGenericMessage(nttMsg.payload);
        NativeTokenTransferCodec.NativeTokenTransfer memory transfer =
            NativeTokenTransferCodec.parseNativeTokenTransfer(genMsg.data);

        return keccak256(NativeTokenTransferCodec.encodeNativeTokenTransfer(transfer));
    }
```

Recommended Mitigation: Consider including the `sourceChainId` when retrieving the digest at `_executeMsg()`. Consider passing `sourceChainId` parameter to `completeInboundQueuedTransfer()` to retrieve the correct digest.

Wormhole: Fixed in commit 399c60a.

Cyfrin: Verified.

7.3 Low Risk

7.3.1 `MultiTokenNtt::overrideLocalAsset` allows overriding native token representations

Description: The `MultiTokenNtt::overrideLocalAsset` lacks validation to ensure that the provided `TokenId` represents a foreign token (from a different chain).

This allows the owner, accidentally or otherwise, to create a local token representation for tokens that are native to the current chain, breaking fundamental protocol invariants. `overrideLocalAsset` only validates that `chainId != 0` but fails to check that `token.chainId != currentChainId`

```
function overrideLocalAsset(TokenId calldata token, address localToken) external onlyOwner {
    require(token.chainId != 0 && token.tokenAddress != bytes32(0));
    // @audit -> MISSING require(token.chainId != chainId)

    TokenMeta memory meta = _queryTokenMetaFromTokenContract(localToken);
    _getLocalTokenStorage()[token.chainId][token.tokenAddress] =
        LocalTokenInfo({token: localToken, meta: meta});
    _getForeignTokenStorage()[localToken] = token;
}
```

This creates an inconsistency in the `getTokenId()` function's mode determination logic:

```
function getTokenId(address token) public view returns (TokenId memory result, Mode mode) {
    result = _getForeignTokenStorage()[token];
    if (result.tokenAddress != bytes32(0)) {
        return (result, Mode.BURNING);  // @audit foreign tokens use BURNING mode
    }
    result.chainId = chainId;
    result.tokenAddress = toWormholeFormat(token);
    return (result, Mode.LOCKING);
}
```

Impact: A token can be registered as both native (LOCKING mode) and foreign (BURNING mode) simultaneously, violating the protocol's core assumption that native tokens are locked/unlocked while foreign representations are minted/burned. Additionally, outbound transfers of the overridden local token will attempt to burn tokens instead of locking them.

Also, if previous transfers have already happened, the totalSupply on Source chain will be locked in `MultiTokenNtt` contract. Shifting to `Burn-and-mint` mode would cause accounting errors to the total supply because of the pre-existing supply locked in the `MultiTokenNtt` contract.

Proof of Concept: Add the following test to `MultiTokenNtt.t.sol`:

```
function testOverrideLocalAsset_NativeToken() public {
    // Setup: Deploy a native token on chain1
    Token nativeToken = new Token();
    nativeToken.initialize("Native Token", "NATIVE", 18);
    uint256 amount = 1000 * 10 ** 18;
    address sender = chain1.addr(user1);
    address recipient = chain2.addr(user1);

    nativeToken.setMinter(address(this));
    nativeToken.mint(sender, amount);

    // Deploy a local variant of native token (doesn't make much sense but just for the sake of testing)
    Token nativeLocalToken = new Token();
    nativeLocalToken.initialize("Malicious", "MAL", 18);
    nativeLocalToken.setMinter(address(this));
    nativeLocalToken.mint(sender, amount);

    nativeLocalToken.setMinter(address(chain1.ntt())); // Allow NTT to mint/burn

    // Owner can override a "native" token representation
```

```
    // @audit This should NOT be allowed as chain1.chainId() == chain1.chainId()
    TokenId memory nativeTokenId = TokenId({
        chainId: chain1.chainId(),
        tokenAddress: toWormholeFormat(address(nativeToken))
    });

    chain1.ntt().overrideLocalAsset(nativeTokenId, address(nativeLocalToken));

    // @audit checks that the mode for native token is locking -> while the local token is burning
    (TokenId memory tokenId1, MultiTokenNtt.Mode mode1) = chain1.ntt().getTokenId(address(nativeToken));
    (TokenId memory tokenId2, MultiTokenNtt.Mode mode2) =
    ↪  chain1.ntt().getTokenId(address(nativeLocalToken));

    // nativeToken correctly returns LOCKING mode
    assertEq(uint(mode1), uint(MultiTokenNtt.Mode.LOCKING), "Native token should be LOCKING");
    assertEq(tokenId1.chainId, chain1.chainId());

    assertEq(uint(mode2), uint(MultiTokenNtt.Mode.BURNING), "Malicious token incorrectly uses BURNING");
    assertEq(tokenId2.chainId, chain1.chainId(), "Both tokens claim same chain!");
    assertEq(tokenId2.tokenAddress, toWormholeFormat(address(nativeToken)), "Malicious maps to
    ↪  native!");

    // Demonstrate the impact: Transfer localNativeToken
    vm.startPrank(sender);
    nativeLocalToken.approve(address(chain1.ntt()), amount);

    // Get balance before transfer
    uint256 balanceBefore = nativeLocalToken.balanceOf(address(chain1.ntt()));

    // This transfer will try to BURN the tokens instead of LOCKING them
    chain1.ntt().transfer(
        MultiTokenNtt.TransferArgs({
            token: address(nativeLocalToken),
            amount: amount,
            recipientChain: chain2.chainId(),
            recipient: toWormholeFormat(recipient),
            refundAddress: toWormholeFormat(sender),
            shouldQueue: false,
            transceiverInstructions: new bytes(1),
            additionalPayload: ""
        })
    );

    // check if tokens were burned instead of locked
    uint256 balanceAfter = nativeLocalToken.balanceOf(address(chain1.ntt()));
    assertEq(balanceAfter, balanceBefore, "Balance should be unchanged if burned");
    assertEq(nativeLocalToken.totalSupply(), 0, "Supply decreased - tokens burned!");


    vm.stopPrank();

}
```

Recommended Mitigation: Consider adding validation that checks chainId to ensure only foreign tokens can have local representations.

```
function overrideLocalAsset(TokenId calldata token, address localToken) external onlyOwner {
    require(token.chainId != 0 && token.tokenAddress != bytes32(0));
    require(token.chainId != chainId, "Cannot override native token representation");

    // Rest of the function...
}
```

Wormhole: Fixed in commit b9cfcad

Cyfrin: Verified.

7.3.2 `MultiTokenNtt::overrideLocalAsset` allows mapping single local token to multiple foreign assets, potentially corrupting internal accounting and rate limiting

Description: The `MultiTokenNtt::overrideLocalAsset` fails to validate whether a local token already represents a different foreign token before creating a new mapping. This allows the owner, accidentally or otherwise, to map a single local token address to multiple different foreign tokens from different chains, creating severe inconsistencies in the protocol's bidirectional token mappings.

The issue arises because:

- `_getLocalTokenStorage()` is a nested mapping that can store multiple foreign→local relationships
- `_getForeignTokenStorage()` is a single mapping that can only store one local→foreign relationship

```
function overrideLocalAsset(TokenId calldata token, address localToken) external onlyOwner {
    require(token.chainId != 0 && token.tokenAddress != bytes32(0));

    TokenMeta memory meta = _queryTokenMetaFromTokenContract(localToken);

    _getLocalTokenStorage()[token.chainId][token.tokenAddress] =
        LocalTokenInfo({token: localToken, meta: meta});

    // @audit Overwrites any existing mapping without validation
    _getForeignTokenStorage()[localToken] = token;
}
```

Impact: Protocol can lose track of which foreign tokens a local token represents. Rate limits can be incorrectly shared between two different assets. Additionally, users trying to bridge local tokens back to their origin might see failed transfers.

Proof of Concept: Add the following to `MultiTokenNtt.t.sol`:

```
function testOverrideLocalAsset_OneLocalMultipleForeignTokens() public {
    address sender = chain1.addr(user1);
    address recipient = chain2.addr(user1);

    // Deploy USDC on chain1
    Token usdcChain1 = new Token();
    usdcChain1.initialize("USDC Chain1", "USDC", 6);
    usdcChain1.setMinter(address(this));
    usdcChain1.mint(sender, 1000 * 10 ** 6); // 1000 USDC

    // Deploy USDT on chain3
    uint16 chain3Id = 3;
    Token usdtChain3 = new Token();
    usdtChain3.initialize("USDT Chain3", "USDT", 6);

    // Deploy a local token on chain2
    Token localToken = new Token();
    localToken.initialize("Local Token", "LT", 18);
    localToken.setMinter(address(chain2.ntt()));

    // Create TokenIds for both foreign tokens
    TokenId memory usdcTokenId = TokenId({
        chainId: chain1.chainId(),
        tokenAddress: toWormholeFormat(address(usdcChain1))
    });

    TokenId memory usdtTokenId = TokenId({
        chainId: chain3Id,
```

```
        tokenAddress: toWormholeFormat(address(usdtChain3))
    });


    // First override: Map localToken to USDC from chain1
    chain2.ntt().overrideLocalAsset(usdcTokenId, address(localToken));

    // Verify the mapping
    address localForUsdc = chain2.ntt().getToken(usdcTokenId);
    assertEq(localForUsdc, address(localToken), "USDC mapped to localToken");

    // Check reverse mapping
    (TokenId memory reverseCheck1, ) = chain2.ntt().getTokenId(address(localToken));
    assertEq(reverseCheck1.chainId, chain1.chainId());
    assertEq(reverseCheck1.tokenAddress, toWormholeFormat(address(usdcChain1)));

    // Second override: Map THE SAME localToken to USDT from chain3
    chain2.ntt().overrideLocalAsset(usdtTokenId, address(localToken));
    console.log("2. Mapped localToken to USDT from chain", chain3Id);

    // Verify both forward mappings still work
    address localForUsdt = chain2.ntt().getToken(usdtTokenId);
    assertEq(localForUsdt, address(localToken), "USDT also mapped to localToken");
    assertEq(localForUsdc, localForUsdt, "Both foreign tokens map to same local!");

    // Check reverse mapping
    (TokenId memory reverseCheck2, ) = chain2.ntt().getTokenId(address(localToken));
    console.log("   Reverse mapping: localToken -> chain", reverseCheck2.chainId);

    // @audit The reverse mapping only points to the LAST assigned foreign token (USDT)
    assertEq(reverseCheck2.chainId, chain3Id, "Now points to USDT, not USDC!");
    assertEq(reverseCheck2.tokenAddress, toWormholeFormat(address(usdtChain3)));

}
```

Recommended Mitigation: Consider adding validation to ensure a local token cannot represent multiple foreign tokens:

```
function overrideLocalAsset(TokenId calldata token, address localToken) external onlyOwner {
    require(token.chainId != 0 && token.tokenAddress != bytes32(0));
    require(token.chainId != chainId, "Cannot override native token representation");

    // @audit Check if localToken already represents a different foreign token
    TokenId memory existing = _getForeignTokenStorage()[localToken];
    if (existing.tokenAddress != bytes32(0)) {
        require(
            existing.chainId == token.chainId &&
            existing.tokenAddress == token.tokenAddress,
            "Local token already represents a different foreign asset"
        );
    }

   // rest of the logic

}
```

Wormhole: Fixed in ceb5d5c

Cyfrin: Verified.

7.3.3 Incorrect error when burning fee token

Description: When using burning mechanism in case there are fee-on-burn tokens, the protocol implements a check to be sure that the amount burned is exactly the same. If not, logic reverts with a `BurnAmountDifferentThanBalanceDiff` error

This error takes two parameters

- `burnAmount`: amount burned
- `balanceDiff`: balance after burning

INttManager.sol#L107-L112

```
    /// @param burnAmount The amount burned.
    /// @param balanceDiff The balance after burning.
    error BurnAmountDifferentThanBalanceDiff(uint256 burnAmount, uint256 balanceDiff);
```

The problem comes with the parameters passed in this custom error, where instead of passing the amount burnt in the first parameter, we pass the amount before burning instead.

NttManager.sol#L433-L435

```
            uint256 balanceBefore = _getTokenBalanceOf(token, address(this));
            ...
            if (mode == Mode.BURNING) {
                {
                    ...
                    uint256 balanceAfterBurn = _getTokenBalanceOf(token, address(this));
                    if (balanceBefore != balanceAfterBurn) {
>>                      revert BurnAmountDifferentThanBalanceDiff(balanceBefore, balanceAfterBurn);
                    }
                }
            }
```

This will result in reverting with incorrect value as we use `balanceBefore` instead of the actual burnt amount in the first parameter

Impact: Incorrect custom error revert message while dealing with fee-on-burn tokens

Recommended Mitigation: Consider making the error type consistent for both MultiTokenNtt and NttManager.

Wormhole: Acknowledged. Will be corrected in future commits.

Cyfrin: Acknowledged.

7.4 Informational

7.4.1 ERC20 tokens not supporting decimals can't be used in NTT

Description: According to EIP20, `decimal()` function is optional. A token can comply with ERX20 standards even without implementing `decimal()` function.

https://eips.ethereum.org/EIPS/eip-20#decimals

> Returns the number of decimals the token uses - e.g. `8`, means to divide the token amount by `100000000` to get its user representation.
>
> OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

In `NttManager::tokenDecimals()` function, we make a static call to `decimal()` function, and in case of failed, we reverted the tx.

NttManager.sol#L646-L655

```
    function tokenDecimals() public view override(INttManager, RateLimiter) returns (uint8) {
        (bool success, bytes memory queriedDecimals) =
            token.staticcall(abi.encodeWithSignature("decimals()"));

        if (!success) {
            revert StaticcallFailed();
        }

        return abi.decode(queriedDecimals, (uint8));
    }
```

Impact: ERC20 tokens not supporting `decimal()` function can't be supported by NttManager

Recommended Mitigation: Consider returning `18`, which is the standard decimal value for the token in case no `decimal()` function exists, this is the method used by OpenZeppelin in ERC4626 implementation.

OpenZeppelin::ERC4626.sol#L79

```
    constructor(IERC20 asset_) {
        (bool success, uint8 assetDecimals) = _tryGetAssetDecimals(asset_);
>>      _underlyingDecimals = success ? assetDecimals : 18;
        _asset = asset_;
    }
```

Wormhole: Acknowledged.

Cyfrin: Acknowledged.

7.4.2 Incorrect invariant documentation in TransceiverRegistry

Description: The `TransceiverRegistry` contract contains incorrect documentation for one of its critical state invariants. The comment at the beginning of the contract states:

```
// 1. If a transceiver is not registered, it should be enabled.
// 2. The value set in the bitmap of transceivers should directly correspond to whether the transceiver
↪  is enabled.
```

The first invariant is logically incorrect and contradicts the actual implementation. According to this statement, unregistered transceivers should be enabled, which is impossible since a transceiver must be registered before it can be enabled.

The actual implementation in `_setTransceiver` correctly enforces that:

- If a transceiver is already registered (but disabled), it re-enables it

- If a transceiver is not registered, it first registers it and then enables it

Recommended Mitigation: Consider changing the invariant to

```
If a transceiver is enabled, it must be registered.
```

Wormhole: Acknowledged.

Cyfrin: Acknowledged.

7.4.3 Unused error types in `MultiTokenNtt.sol`

Description: Following error types are defined but never used.

`InvalidSender()` `InvalidTargetChain(uint16 targetChain, uint16 chainId)` `FailedToDeployToken()`

Recommended Mitigation: Consider removing error types that are unused and/or used elsewhere.

Wormhole: Acknowledged.

Cyfrin: Acknowledged.

7.4.4 Incorrect inline comments for `TransceiverAdded` event

Description: `TransceiverAdded` event has inline comment specific to `TransceiverRemoved` event:

```
//IManagerBase.sol
 /// @notice Emitted when an transceiver is removed from the nttManager. <<<<<
    /// @dev Topic0
    ///      0xf05962b5774c658e85ed80c91a75af9d66d2af2253dda480f90bce78aff5eda5.
    /// @param transceiver The address of the transceiver.
    /// @param transceiversNum The current number of transceivers.
    /// @param threshold The current threshold of transceivers.
    event TransceiverAdded(address transceiver, uint256 transceiversNum, uint8 threshold);
```

Recommended Mitigation: Consider correcting the comments.

Wormhole: Acknowledged.

Cyfrin: Acknowledged.