CANTINA

Wormhole native token transfers

Competition

April 23, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Wormhole is the leading interoperability platform powering multichain applications and bridges at scale. The platform is designed to connect different blockchains, allowing for the secure and efficient transfer of data and assets.

From Mar 13th to Mar 27th Cantina hosted a competition based on native-token-transfers. The participants identified a total of **29** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 2
- Low Risk: 18
- Gas Optimizations: 0
- Informational: 9

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 Transfers will fail if there's a transceiver with index greater than or equal to the number of enabled transceivers

Submitted by neumo, also found by Gerard Persoon, gjaldon, 0x3b, bin2chen, 0xhuy0512, elhaj and trachev

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Calls to `quoteDeliveryPrice` will revert if the `index` of any of the enabled transceivers passed as parameter is greater than or equal to the `transceiverInstructions` array length.

This means that, for instance, if we pass an array of 5 instructions, there cannot be any transceiver enabled with index 5 or greater, otherwise the call will fail. The function is defined in the `ManagerBase` contract as follows:

```
// ...
function quoteDeliveryPrice(
    uint16 recipientChain,
    TransceiverStructs.TransceiverInstruction[] memory transceiverInstructions,
    address[] memory enabledTransceivers
) public view returns (uint256[] memory, uint256) {
    uint256 numEnabledTransceivers = enabledTransceivers.length;
    mapping(address => TransceiverInfo) storage transceiverInfos = _getTransceiverInfosStorage();

    uint256[] memory priceQuotes = new uint256[](numEnabledTransceivers);
    uint256 totalPriceQuote = 0;
    for (uint256 i = 0; i < numEnabledTransceivers; i++) {
        address transceiverAddr = enabledTransceivers[i];
        uint8 registeredTransceiverIndex = transceiverInfos[transceiverAddr].index;
        uint256 transceiverPriceQuote = ITransceiver(transceiverAddr).quoteDeliveryPrice(
            recipientChain, transceiverInstructions[registeredTransceiverIndex]
        );
        priceQuotes[i] = transceiverPriceQuote;
        totalPriceQuote += transceiverPriceQuote;
    }
    return (priceQuotes, totalPriceQuote);
}
// ...
```

Inside the loop, the transceiver index is retreived:

```
uint8 registeredTransceiverIndex = transceiverInfos[transceiverAddr].index;
```

And then the function calls the individual `quoteDeliveryPrice` of the transceiver:

```
ITransceiver(transceiverAddr).quoteDeliveryPrice(
    recipientChain, transceiverInstructions[registeredTransceiverIndex]
);
```

So, if the index is greater than or equal to the `transceiverInstructions` array length, the call will revert:

```
transceiverInstructions[registeredTransceiverIndex]
```

Function `quoteDeliveryPrice` of `ManagerBase` contract is called in:

- `_prepareForTransfer` of `ManagerBase` contract

Where the `enabledTransceivers` are computed as:

```
address[] memory enabledTransceivers = _getEnabledTransceiversStorage();
```

So all enabled transceivers in the `NttManager` are passed to `quoteDeliveryPrice`.

Also, the `bytes memory transceiverInstructions` passed as parameter are parsed to get the `TransceiverStructs.TransceiverInstruction[]` array used in `quoteDeliveryPrice`.

```
instructions = TransceiverStructs.parseTransceiverInstructions(
    transceiverInstructions, numEnabledTransceivers
);
```

Function `parseTransceiverInstructions` in `TransceiverStructs` library is defined as follows:

```
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;
}
```

We can see the length of the instructions array returned is always the number of enabled transceivers.

`_prepareForTransfer` is called in:

- `_transfer` in `NttManager` contract

Which in turn is called in:

- `completeOutboundQueuedTransfer` in `NttManager` contract
- `_transferEntryPoint` in `NttManager` contract

`_transferEntryPoint` is called in:

- `transfer(uint256 amount, uint16 recipientChain, bytes32 recipient)` in `NttManager` contract
- `transfer(uint256 amount, uint16 recipientChain, bytes32 recipient, bool shouldQueue, bytes memory transceiverInstructions)` in `NttManager` contract

This means any call to `completeOutboundQueuedTransfer` and to both versions of `transfer` will fail as long as there is at least one enabled transceiver which index is greater than or equal to the number of enabled transceivers.

Recommendation: The fix should be made in file `parseTransceiverInstructions`, because that's the one responsible for returning the array of instructions, and is currently returning an array of size `numEnabled-Transceivers`:

```
  function parseTransceiverInstructions(
      bytes memory encoded,
-      uint256 numEnabledTransceivers
+      uint256 numRegisteredTransceivers
  ) 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);
+          new TransceiverInstruction[](numRegisteredTransceivers);
```

And then, in the call to `parseTransceiverInstructions` from `ManagerBase`'s `_prepareForTransfer` function, we should pass the number of registered transceivers instead of only the number of enabled transceivers:

```
  instructions = TransceiverStructs.parseTransceiverInstructions(
-     transceiverInstructions, numEnabledTransceivers
+     transceiverInstructions, _getRegisteredTransceiversStorage().length
  );
```

Proof of concept: The following test (which you can add to the project's `evm/test/NttManager.t.sol` file) shows how, after having two transceivers registered and removing the first one, a call to transfer reverts because the index of the enabled transceiver is 1, which is equal to the number of enabled transceivers and provokes an Array OOB error.

```
function test_transceiverInstructionsOutOfBounds() public {

    DummyTransceiver dt1 = new DummyTransceiver(address(nttManager));
    nttManager.setTransceiver(address(dt1));

    nttManager.removeTransceiver(address(dummyTransceiver));

    address user_A = address(0x123);
    address user_B = address(0x456);

    DummyToken token = DummyToken(nttManager.token());

    uint8 decimals = token.decimals();

    nttManager.setPeer(chainId, toWormholeFormat(address(0x1)), 9, type(uint64).max);
    nttManager.setOutboundLimit(packTrimmedAmount(type(uint64).max, 8).untrim(decimals));

    token.mintDummy(address(user_A), 5 * 10 ** decimals);

    vm.startPrank(user_A);

    token.approve(address(nttManager), 3 * 10 ** decimals);

    nttManager.transfer(
        1 * 10 ** decimals, chainId, toWormholeFormat(user_B), false, new bytes(1)
    );
}
```

3.1.2 Cross-chain transactions can be replayed when the chain undergoes a hard fork

Submitted by pks271, also found by J4X98

Severity: Medium Risk

Context: NttManager.sol#L234-L251

Description: During a hard fork, if `InboundQueued` has still pending transactions, these can be replayed on another chain. The root cause is that the `NttManager.completeInboundQueuedTransfer` function lackz `checkFork(evmChainId)` validation like the `executeMsg` function.

Although the chances are low, the impact is high as malicious users can extract grat profitz. See the omni bridge vulnerability during Ethereum's transition to a proof of stake consensus mechanism.

Recommendation: Add `checkFork(evmChainId)` check to `completeInboundQueuedTransfer` function.