Thermae Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

0kage

January 10, 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

Thermae is a cross-chain liquidity protocol which uses Wormhole and Uniswap V3 to implement cross-chain ERC20 swaps. There are 2 primary transactions: an initiating transaction on the source chain and a receiving transaction on the destination chain. Users can offer a relayer fee to have a relayer perform the receiving transaction for them. The only admin functionality is the ability to set an address to receive the relayer fees.

5 Audit Scope

Following contracts were included in the scope for this audit:

```
contracts/IERC20.sol
contracts/ITokenBridge.sol
contracts/IWETH.sol
contracts/IWormhole.sol
contracts/Portico.sol
contracts/PorticoStructs.sol
```

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the Thermae smart contracts provided by Wormhole / GFX Labs. In this period, a total of 12 issues were found.

The findings consist of 1 High, 2 Medium & 1 Low severity issues with the remainder being informational and gas optimizations. The only High issue could result in users receiving less bridged tokens than expected due to swaps being exploited by MEV attackers. One medium issue would prevent the protocol working with some non-standard

ERC20 tokens and the other medium issue could result in the relayer being unable to retrieve bridged funds and users potentially receiving less bridged tokens than expected.

Additionally stateless fuzz testing was performed to provide greater assurance that the Typescript flag encoding and the Solidity flag decoding worked together correctly for a wide range of possible inputs.

All of the issues were successfully mitigated. The mitigations resulted in the deletion of significant amounts of code and greater simplification of the contract, reducing the potential attack surface while increasing the contract's robustness.

Summary

Project Name	Thermae
Repository	gfx-wormhole-sneak-peak
Commit	fb5b49e78b40...
Audit Timeline	Feb 1st - Feb 14th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	2
Low Risk	1
Informational	5
Gas Optimizations	3
Total Issues	12

Summary of Findings

[H-1] On-chain slippage calculation using exchange rate derived from `pool.slot0` can be easily manipulated	Resolved
[M-1] Checking `bool` return of ERC20 `approve` and `transfer` breaks protocol for mainnet USDT and similar tokens which don't return true	Resolved
[M-2] No precision scaling or minimum received amount check when subtracting `relayerFeeAmount` can revert due to underflow or return less tokens to user than specified	Resolved
[L-1] Use low level `call()` to prevent gas griefing attacks when returned data not required	Resolved
[I-1] Missing sanity check for address validity in `PorticoBase::unpadAddress`	Resolved
[I-2] Move payable `receive()` function from `PorticoBase` into `PorticoFinish`	Resolved
[I-3] `Portico::start` not used internally could be marked external	Resolved

[I-4] `TokenBridge::isDeployed` could be declared pure	Resolved
[I-5] Remove unused code	Resolved
[G-1] Fail fast in `_completeTransfer` by checking for incorrect address/chainId immediately after calling `TOKENBRIDGE.parseTransferWithPayload`	Resolved
[G-2] Don't initialize variables with default value	Resolved
[G-3] Use custom errors instead of revert error strings	Resolved

7 Findings

7.1 High Risk

7.1.1 On-chain slippage calculation using exchange rate derived from `pool.slot0` can be easily manipulated

Description: On-chain slippage calculation using price from `pool.slot0` can be easily manipulated causing users to receive less tokens than they intended.

Impact: Swaps can result in users receiving less tokens than they intended.

Proof of Concept: `Portico::calcMinAmount` attempts to on-chain calculate the minimum amount of tokens a swap should return. It does this using:

1) L85 taking as input either the `maxSlippageStart` or `maxSlippageFinish` parameters which users can specify for the 2 possible swaps,
2) L135 getting the current exchange rate on-chain by reading price information from `pool.slot0`

The problem is that `pool.slot0` is easy to manipulate using flash loans so the actual exchange rate used in the slippage calculation could be far worse than what the user expects; it is very likely users will be continually exploited via sandwich attacks on the swaps.

Recommended Mitigation:

1. If price information is required on-chain, use Uniswap V3 TWAP instead of `pool.slot0` for more manipulation-resistant price info (note: this does not offer the same level of protection on Optimism),
2. Use `minAmountReceivedStart` and `minAmountReceivedFinish` parameters instead of `maxSlippageStart` and `maxSlippageFinish` and remove the on-chain slippage calculation. There is no "safe" way to calculate slippage on-chain. If users specify % slippage params, calculate the exact minimum amounts off-chain and pass these in as input.

Wormhole: Fixed in commit af089d6.

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 Checking `bool` return of ERC20 `approve` and `transfer` breaks protocol for mainnet USDT and similar tokens which don't return true

Description: Checking `bool` return of ERC20 `approve` and `transfer` breaks protocol for mainnet USDT and similar tokens which don't return true even though the calls were successful.

Impact: Protocol won't work with mainnet USDT and similar tokens.

Proof of Concept: Portico.sol L58, 61, 205, 320, 395, 399.

Recommended Mitigation: Use SafeERC20 or SafeTransferLib.

Wormhole: Fixed in commits 3f08be9 & 55f93e2.

Cyfrin: Verified.

7.2.2 No precision scaling or minimum received amount check when subtracting `relayerFeeAmount` can revert due to underflow or return less tokens to user than specified

Description: `PorticoFinish::payOut` L376 attempts to subtract the `relayerFeeAmount` from the final post-bridge and post-swap token balance:

```
finalUserAmount = finalToken.balanceOf(address(this)) - relayerFeeAmount;
```

There is no precision scaling to ensure that `PorticoFinish`'s token contract balance and `relayerFeeAmount` are in the same decimal precision; if the `relayerFeeAmount` has 18 decimal places but the token is USDC with only 6 decimal places, this can easily revert due to underflow resulting in the bridged tokens being stuck.

An excessively high `relayerFeeAmount` could also significantly reduce the amount of post-bridge and post-swap tokens received as there is no check on the minimum amount of tokens the user will receive after deducting `relayerFeeAmount`. This current configuration is an example of the "MinTokensOut For Intermediate, Not Final Amount" vulnerability class; as the minimum received tokens check is before the deduction of `relayerFeeAmount` a user will always receive less tokens than their specified minimum if `relayerFeeAmount > 0`.

Impact: Bridged tokens stuck or user receives less tokens than their specified minimum.

Recommended Mitigation: Ensure that token balance and `relayerFeeAmount` have the same decimal precision before combining them. Alternatively check for underflow and don't charge a fee if this would be the case. Consider enforcing the user-specified minimum output token check again when deducting `relayerFeeAmount`, and if this would fail then decrease `relayerFeeAmount` such that the user at least receives their minimum specified token amount.

Another option is to check that even if it doesn't underflow, that the remaining amount after subtracting `relayerFeeAmount` is a high percentage of the bridged amount; this would prevent a scenario where `relayerFeeAmount` takes a large part of the bridged amount, effectively capping `relayerFeeAmount` to a tiny % of the post-bridge and post-swap funds. This scenario can still result in the user receiving less tokens than their specified minimum however.

From the point of view of the smart contract, it should protect itself against the possibility of the token amount and `relayerFeeAmount` being in different decimals or that `relayerFeeAmount` would be too high, similar to how for example L376 inside `payOut` doesn't trust the bridge reported amount and checks the actual token balance.

Wormhole: Fixed in commit 05ba84d by adding an underflow check. Any misbehavior is due to bad user input and should be corrected off-chain. Only the user is able to set the relayer fee in the input parameters.

Cyfrin: Verified potential underflow due to mismatched precision between relayer fee & token amount is now handled. The implementation now favors the relayer however this is balanced by the fact that only the user can set the relayer fee, so the attack surface is limited to self-inflicted harm. If in the future another entity such as the relayer could set the relayer fee then this could be used to drain the bridged tokens, but with the current implementation this is not possible unless the user sets an incorrectly large relayer fee which is self-inflicted.

7.3 Low Risk

7.3.1 Use low level `call()` to prevent gas griefing attacks when returned data not required

Description: Using `call()` when the returned data is not required unnecessarily exposes to gas griefing attacks from huge returned data payload. For example:

```
(bool sentToUser, ) = recipient.call{ value: finalUserAmount }("");
require(sentToUser, "Failed to send Ether");
```

Is the same as writing:

```
(bool sentToUser, bytes memory data) = recipient.call{ value: finalUserAmount }("");
require(sentToUser, "Failed to send Ether");
```

In both cases the returned data will be copied into memory exposing the contract to gas griefing attacks, even though the returned data is not used at all.

Impact: Contract unnecessarily exposed to gas griefing attacks.

Recommended Mitigation: Use a low-level call when the returned data is not required, eg:

```
bool sent;
assembly {
    sent := call(gas(), recipient, finalUserAmount, 0, 0, 0, 0)
}
if (!sent) revert Unauthorized();
```

Consider using ExcessivelySafeCall.

Wormhole: Fixed in commit 5f3926b.

Cyfrin: Verified.

7.4 Informational

7.4.1 Missing sanity check for address validity in `PorticoBase::unpadAddress`

Description: `PorticoBase::unpadAddress` is a re-implementation of `Utils::fromWormholeFormat` from the Wormhole Solidity SDK, but is missing a sanity check for address validity which is in the SDK implementation.

Recommended Mitigation: Consider adding the address validity sanity check to `PorticoBase::unpadAddress`.

Wormhole: Fixed in commit 6208dd1.

Cyfrin: Verified.

7.4.2 Move payable `receive()` function from `PorticoBase` into `PorticoFinish`

Description: Move payable `receive()` function from `PorticoBase` into `PorticoFinish` since `PorticoFinish` is the only contract which needs to receive eth when it calls `WETH.withdraw()`.

`PorticoStart` which also inherits from `PorticoBase` never needs to receive eth apart from the payable `start` function, so does not need to have or inherit a payable `receive()` function.

Wormhole: Fixed in commit 6208dd1.

Cyfrin: Verified.

7.4.3 `Portico::start` not used internally could be marked external

Description: `Portico::start` not used internally could be marked external.

Wormhole: Fixed in commit 6208dd1.

Cyfrin: Verified.

7.4.4 `TokenBridge::isDeployed` could be declared pure

Description: `TokenBridge::isDeployed` could be declared pure. Also not sure what the point of this contract is; if it is used for testing perhaps move it into a `mocks` directory.

Wormhole: Removed this contract.

Cyfrin: Verified.

7.4.5 Remove unused code

Description:

File: PorticoStructs.sol L67-79:

```
  //16 + 32 + 24 + 24 + 16 + 16 + 8 + 8 == 144
  struct packedData {
    uint16 recipientChain;
    uint32 bridgeNonce;
    uint24 startFee;
    uint24 endFee;
    int16 slipStart;
    int16 slipEnd;
    bool wrap;
    bool unwrap;
  }
```

Wormhole: Fixed in commit 6208dd1.

Cyfrin: Verified.

7.5 Gas Optimization

7.5.1 Fail fast in `_completeTransfer` by checking for incorrect address/chainId immediately after calling `TOKENBRIDGE.parseTransferWithPayload`

Description: Fail fast in `_completeTransfer` by checking for incorrect address/chainId immediately after calling `TOKENBRIDGE.parseTransferWithPayload` per the example code.

Impact: Gas optimization; want to fail fast instead of performing a number of unnecessary operations then failing later anyway.

Proof of Concept: Portico.sol L278-300.

Recommended Mitigation: Perform the L300 check immediately after L278.

Wormhole: Fixed in commit 5f3926b.

Cyfrin: Verified.

7.5.2 Don't initialize variables with default value

Description: Don't initialize variables with default value, eg in `TickMath::getTickAtSqrtRatio()`:

```
uint256 msb = 0;
```

Impact: Gas optimization.

Wormhole: `TickMath` is no longer used as on chain slippage calculations are not being done anymore.

7.5.3 Use custom errors instead of revert error strings

Description: Using custom errors instead of revert error strings to reduce deployment and runtime cost:

```
File: Portico.sol

64:         require(token.approve(spender, 0), "approval reset failed");

67:       require(token.approve(spender, 2 ** 256 - 1), "infinite approval failed");

185:     require(poolExists, "Pool does not exist");

215:       require(value == params.amountSpecified + whMessageFee, "msg.value incorrect");

225:       require(value == whMessageFee, "msg.value incorrect");

232:       require(params.startTokenAddress.transferFrom(_msgSender(), address(this),
↪  params.amountSpecified), "transfer fail");

240:       require(amount >= params.amountSpecified, "transfer insufficient");

333:     require(unpadAddress(transfer.to) == address(this) && transfer.toChain == wormholeChainId,
↪  "Token was not sent to this address");

420:         require(sentToUser, "Failed to send Ether");

425:         require(sentToRelayer, "Failed to send Ether");

432:         require(finalToken.transfer(recipient, finalUserAmount), "STF");

436:         require(finalToken.transfer(feeRecipient, relayerFeeAmount), "STF");
```

Wormhole: Error strings have all been confirmed to be length < 32, this is sufficient for the purposes of this contract.