Zellic

Hourglass

Smart Contract Security Assessment

September 26, 2023

Prepared for:

Sam Trautwein

Tristero Incorporated

Prepared by:

Will Bowling, Aaron Esau, and Junyi Wang

Zellic Inc.

Contents

About Zellic 3

1 Executive Summary 4

1.1 Goals of the Assessment . . . 4

1.2 Non-goals and Limitations . . . 4

1.3 Results . . . 5

2 Introduction 6

2.1 About Hourglass . . . 6

2.2 Security Evaluation . . . 6

2.3 Methodology . . . 6

2.4 Scope . . . 8

2.5 Project Overview . . . 8

2.6 Project Timeline . . . 8

3 Detailed Findings 10

3.1 State updated in the incorrect order . . . 10

3.2 Configuration centralization risks . . . 17

3.3 Erroneous check for max taker orders . . . 18

3.4 Wrong `lz_cid` results in locked funds . . . 19

3.5 Lack of maker order-cancellation mechanism . . . 20

3.6 Missing result check in `transferFrom` helper . . . 21

3.7 Limited number of maker orders can be placed . . . 23

3.8 Off-by-one `epochspan` check . . . 25

3.9 Hardcoded minimum order size . . . 27

4 Discussion **28**

4.1 Consider making `lzc` immutable 28

4.2 Lack of fees 28

5 Threat Model **29**

5.1 Module: Multichain.sol 29

6 Assessment Results **39**

6.1 Disclaimer 39

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Tristero Incorporated from August 30th to September 12th. During this engagement, Zellic reviewed Hourglass's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can a malicious user cause an order to be erroneously deleted, resulting in stuck funds?
- Is it possible for a specially crafted series of orders to cause a reversion on the destination chain, resulting in stuck funds?
- Can funds be stolen from the contract?
- Can a race condition between when a destination LayerZero endpoint sends a message to a source endpoint, and the source to the destination endpoint, cause integrity issues?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- LayerZero or smart contract–deployment configuration
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, the limited amount of time and lack of comprehensive testing (e.g., of `roll_taker_orders` and its interactions with other code) prevented us from fully assessing the protocol with dynamic analysis.

Due to the protocol's complexity, we strongly recommend creating comprehensive end-to-end tests that achieve 100% branch coverage using Foundry or another common, maintainable testing framework. Additionally, we suggest adding detailed in-line code documentation for each branch, explaining its purpose and any edge

cases it handles. This will greatly enhance the code's comprehensibility and maintainability. Please see section 2.2 for our full recommendations.

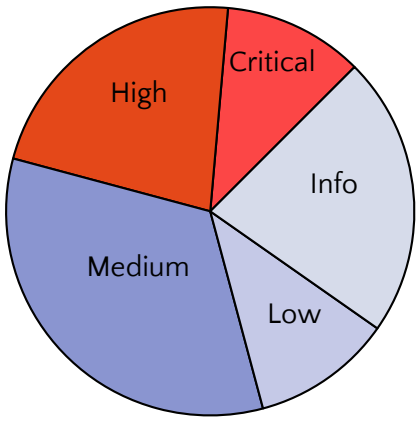
1.3 Results

During our assessment on the scoped Hourglass contracts, we discovered nine findings. One critical issue was found. Two were of high impact, three were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Tristero Incorporated's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	2
Medium	3
Low	1
Informational	2



2 Introduction

2.1 About Hourglass

Hourglass is a decentralized protocol engineered to streamline and secure the transfer of stablecoins across multiple blockchain networks. With Hourglass, you can effortlessly swap one type of stablecoin for another, even if they reside on different blockchains.

2.2 Security Evaluation

Due to the protocol's high complexity, and the limited time allocated to this engagement, we cannot be 100% sure of the security of the protocol at this time. We highly recommend seeking third-party re-audits of the code from separate firms to ensure it functions as intended. Until then, we recommend users exercise caution when interacting with the protocol.

Given the high complexity, the protocol's testing and documentation should be made more thorough before launch. We strongly recommend creating comprehensive end-to-end tests that achieve 100% branch coverage using Foundry or another common, maintainable testing framework. Additionally, we suggest adding detailed in-line code documentation for each branch, explaining its purpose and any edge cases it handles. This will greatly enhance the code's comprehensibility and maintainability.

2.3 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.4 Scope

The engagement involved a review of the following targets:

Hourglass Contracts

Repository	https://github.com/tristeroresearch/Hourglass_Contracts
Version	Hourglass_Contracts: `60733e6e8d2a8f14be8ff62f10641f10766fa3d6`
Program	Spoke
Type	Solidity
Platform	EVM-compatible

2.5 Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of four person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Will Bowling, Security Engineer
vakzz@zellic.io

Aaron Esau, Security Engineer
aaron@zellic.io

Junyi Wang, Security Engineer
junyi@zellic.io

2.6 Project Timeline

The key dates of the engagement are detailed below.

August 30, 2023	Start of primary review period
August 31, 2023	Kick-off call
September 12, 2023	End of primary review period
September 13, 2023	Draft report delivered

3 Detailed Findings

3.1 State updated in the incorrect order

- **Target**: Spoke
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

In the following code, the payout should happen after `resolve_epoch`.

```
//Payout the orders
payout_orders(sell_token, payload.orders,
    selected_pair.sums.maker_default_quantity);

if (qd>0) {
    roll_taker_orders(sell_token, buy_token, lz_cid, qd);
}

//**IF NEEDED: BOUNCE BACK A LZ MESSAGE
if (!selected_pair.isAwaiting) {
    //RESOLVE THE PAIR
    require(address(this).balance ≥ MINGAS, "!gasLimit bounce");
    resolve_epoch(sell_token, buy_token, lz_cid);
}
```

Impact

Below, we list at least two potential impacts of this bug.

Theft of funds by abusing defaulting makers

A spoke is a node on a chain that interacts with other spokes on other chains using LayerZero's cross-chain messaging protocol. Each spoke handles placing orders on its chain, sending the orders to other chains, receiving orders from other chains, and calculating/performing payouts.

When there is an inbound LayerZero message for a spoke, the first thing that happens

is that any orders in the payload are transferred out in the `payout_orders` function. After that, the current epoch is resolved using `resolve_epoch`, which will calculate any payouts for the other spoke and pull funds from the makers.

```
uint96 qd=payload.default_quantity;

//Payout the orders
payout_orders(sell_token, payload.orders,
    selected_pair.sums.maker_default_quantity);

if (qd>0) {
    roll_taker_orders(sell_token, buy_token, lz_cid, qd);
}


//**IF NEEDED: BOUNCE BACK A LZ MESSAGE
if (!selected_pair.isAwaiting) {
    //RESOLVE THE PAIR
    require(address(this).balance ≥ MINGAS, "!gasLimit bounce");
    resolve_epoch(sell_token, buy_token, lz_cid);
}

//Store new sums
selected_pair.sums.contra_taker_sum=payload.taker_sum;
selected_pair.sums.contra_maker_sum=payload.maker_sum;

//unlock the contract
selected_pair.isAwaiting=false;
```

The issue is that the payouts are happening before the funds have been pulled from the makers and there is no guarantee that the makers will not default and be unable to fund the original transfer.

If the spoke does not have enough token balance to cover the initial transfer for the payout, the taker will not receive any tokens on the destination chain, the order will be resolved, and the tokens will be stuck on the spoke.

If the spoke does have enough balance to cover the initial transfer and the maker defaults, the taker will receive the tokens on the destination chain and the order will be kept open, allowing it to be fulfilled again or expire and have the original funds returned.

An attacker could abuse this bug by creating the appropriate taker and defaulting

maker orders to steal any collateral currently being held by the spoke.

Funds being stuck in the chain

The following loop in the `send_orders` function's purpose is to match taker orders:

```
(quantities.taker_demand, quantities.maker_demand)
    = get_demands(selected_pair.sums.taker_sum,
    selected_pair.sums.maker_sum, selected_pair.sums.contra_taker_sum,
    selected_pair.sums.contra_maker_sum, quant_default);

// [...]
while (quantities.taker_demand>0) {
    //load order
    order_amount = taker_orders[quantities.i].amount
    < quantities.taker_demand ? taker_orders[quantities.i].amount
    : quantities.taker_demand;
    order_sender=taker_orders[quantities.i].sender;
    order_next=taker_orders[quantities.i].next;

    //append order
    Payout memory newPayout = Payout({
        sender: order_sender,
        amount: order_amount
    });

    orders_to_send[quantities.index]=newPayout;
    quantities.index+=1;


    //End Conditions
    if (quantities.taker_demand==order_amount){
        selected_pair.index.taker_capital=order_amount;
        selected_pair.index.taker_sent=quantities.i;

    selected_pair.index.taker_amount=taker_orders[quantities.i].amount;


        if (order_amount==taker_orders[quantities.i].amount){
            quantities.i=order_next;
        }
```

```
        else {
            taker_orders[quantities.i].amount-=order_amount;
        }
        quantities.taker_demand=0;
    }

    else{
        quantities.i=order_next;
        quantities.taker_demand-=order_amount;
    }
}
```

It depends on the proper calculation of `quantities.taker_demand` to know when to stop walking up the `taker_orders` linked list. That value comes from the `get_demands` function, which uses the sums accounted for during the previous epoch, which gets stored in `resolve_epoch` after `send_orders`:

```
//(N-1) Step 1
(Payout[] memory orders_to_send, uint96 quantity_default)
    = send_orders(sell_token, buy_token, lz_cid, 0);

// [...]

//Update the sums
selected_pair.sums.taker_sum=taker_sum;
selected_pair.sums.maker_sum=maker_sum;
```

When the sums are outdated, there are two possibilities:

1. **The taker sum is too low.** This is not necessarily bad. For example, if a taker order is placed on the spoke, the sums will not be updated until the next epoch. Those orders simply will not be matched because the while loop in `send_orders` will not walk far enough down the linked list.

2. **The taker sum is higher than it should be.** This is bad. The `taker_demand` in `send_orders` will be erroneously high, causing the while loop in `send_orders` to repeat more times than it should.

For the taker sum to be too high, the actual sum of taker order amounts — by the time `get_demands` is called in `send_orders` — must be lower than the expected sum that is

calculated in the previous epoch. That means that at least one order amount must decrease without the sums being updated.

At least one place where the order amount can decrease before `send_orders` during the handling of a LayerZero message is in the `roll_taker_orders` function.

Per the in-code documentation, the function serves the following purpose:

> [The] function is used when this spoke's taker orders were sent to be distributed at the opposite spoke, but some or all opposite spoke's makers failed to fund.

In the function, there is a while loop that identifies how many orders that maker-defaulted funds would have filled. If there is any remaining `qd`, it partially fills the next order in the if condition:

```
while (qd>order.amount){

    roll=true;
    order.epoch=current_epoch;
    qd-=order.amount;

    //go to the prior order
    i=taker_orders[i].prev;
    order=taker_orders[i];

}

if (qd>0){
    roll=true;
    order.amount=qd;
    order.epoch=current_epoch;
    qd=0;
}
```

This means that the `order.amount` can decrease before the `send_orders` call during a `_nonblockingLzReceive` call; that is, the expected taker sum can be too high.

If the expected taker sum is too high in `send_orders`, the `taker_demand` will be mistakenly high — which causes the aforementioned while loop to iterate too many times, potentially resulting in an index-out-of-bounds reversion.

If a LayerZero message cannot be executed — which is the case when the execution

reverts — funds may not be paid out and may be permanently stuck in the smart contract.

We provided a proof-of-concept to Tristero Incorporated to demonstrate this:

```
    |   |   |   ├─ [83799] spokeSrc::nonblockingLzReceive(2, 0xa4ad4f68d0b91cfd19687c881e50f3a00242828c1d1499e62
2d69689cdf9004d05ec547d650ff211, 2, 0x0000000000000000000000000000000000000000000000000000000000000020000000000000
0000000000005991a2df15a8f6a256d3ec51e99254cd3fb576a900000000000000000000005615deb798bb3e4dfa0139dfa1b3d433cc
23b72f0000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000254582f4000000000000000000000000000000000000000000
0000000000000000000000000000000010000000000000000000000000000000000000000000000000000000000129b9a6c00000000000
0000000000000000000000000000000000000000000000010000000000000000000000000000000000000000000000000000000000000000
0000030000000000000000000000000000000000000000000000000000000b0000000000000000000000000000000000000000000000000
00000000000012a05f2000000000000000000000000000000000000000000000000000000000000000b0000000000000000000000000000
0000000000000000000000000012a05f2000000000000000000000000000000000000000000000000000000000b0000000000000000000
0000000000000000000000000000000000000000000004c4b40)
    |   |   |   |   ├─ emit ResponseStatus(canResolve: true, payload_epoch: 1, this_spoke_epoch: 1, this_timesta
mp: 123)
    |   |   |   |   ├─ [25038] X::transfer(walletB: [0x000000000000000000000000000000000000000b], 5000000 [5e6])

    |   |   |   |   |   ├─ emit Transfer(from: spokeSrc: [0x1d1499e622D69689cdf9004d05Ec547d650Ff211], to: walle
tB: [0x000000000000000000000000000000000000000b], value: 5000000 [5e6])
    |   |   |   |   |   └─ ← true
    |   |   |   |   ├─ [3138] X::transfer(walletB: [0x000000000000000000000000000000000000000b], 5000000000 [5e9
])
    |   |   |   |   |   ├─ emit Transfer(from: spokeSrc: [0x1d1499e622D69689cdf9004d05Ec547d650Ff211], to: walle
tB: [0x000000000000000000000000000000000000000b], value: 5000000000 [5e9])
    |   |   |   |   |   └─ ← true
    |   |   |   |   ├─ [3138] X::transfer(walletB: [0x000000000000000000000000000000000000000b], 5000000000 [5e9
])
    |   |   |   |   |   ├─ emit Transfer(from: spokeSrc: [0x1d1499e622D69689cdf9004d05Ec547d650Ff211], to: walle
tB: [0x000000000000000000000000000000000000000b], value: 5000000000 [5e9])
    |   |   |   |   |   └─ ← true
    |   |   |   |   └─ ← "Index out of bounds"
    |   |   |   ├─ emit MessageFailed(_srcChainId: 2, _srcAddress: 0xa4ad4f68d0b91cfd19687c881e50f3a00242828c1d1
499e622d69689cdf9004d05ec547d650ff211, _nonce: 2, _payload: 0x0000000000000000000000000000000000000000000000000
0000000000020000000000000000000000005991a2df15a8f6a256d3ec51e99254cd3fb576a900000000000000000000005615deb798
bb3e4dfa0139dfa1b3d433cc23b72f0000000000000000000000000000000000000000000000000000000000000020000000000000000000
0000000000000000000000000000000000000000000000000000000000000254582f4000
0000000000000000000000000000000000000000000000010000000000000000000000000000000000000000000000000000000000000000
00000129b9a6c0000000000000000000000000000000000000000000000000000000000000010000000000000000000000000000000000
```

Recommendations

The payouts should occur after the epoch has been resolved to ensure that the correct amount of funds is available.

Remediation

This issue has been acknowledged by Tristero Incorporated, and a fix was implemented in commit e7f944d3.

Below is Tristero Incorporated's response:

> Two weeks before the audit, two lines of code were moved earlier in the `receive()` function. It's visible in the fix commit hash that the only modification was moving the two lines back to their original position. Also, note that our test cases have a test to catch this behavior, we just were running short on time and didn't run them on the commit hash we sent to zellic.

3.2 Configuration centralization risks

- **Target**: Spoke
- **Category**: Protocol Risks
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Medium

Description

Note that the contract owner can change the LayerZero User Agent (UA) configuration — the settings that determine the number of block confirmations required for cross-chain messages, which relayer and oracle to use, and more — at any time, including adding a new trusted remote in the endpoint (e.g., using `setspoke` or one of the LzApp configuration functions).

Impact

A contract owner can potentially configure a malicious trusted remote that spoofs an array of `Payout` structs, thereby draining funds from the contract.

Recommendations

Ensure the owner is a governance contract, or otherwise accept the centralization risks.

Remediation

This issue has been acknowledged by Tristero Incorporated.

3.3 Erroneous check for max taker orders

- **Target**: Spoke
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The following line of code is the check in the `placeTaker` function that ensures the maximum number of open taker orders is not exceeded:

```
require((selected_pair.index.taker_tail-selected_pair.index.taker_head)
    < max_orders, "!takerStack");
```

However, the distance between indexes of the head and tail of the linked list does not necessarily equal the number of open orders in the linked list, since orders in between the indexes could be closed/unlinked.

Impact

The `!takerStack` reversion may happen more frequently than expected — potentially with only two orders being open.

Recommendations

Change the check to this:

```
require(getTakers(sell_token, buy_token, lz_cid).length ≤ max_orders,
    "!takerStack");
```

Remediation

Tristero Incorporated noted that this behavior is intended.

3.4 Wrong `lz_cid` results in locked funds

- **Target**: Spoke
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

The `placeMaker` and `placeTaker` functions both take an `lz_cid` argument, which is a unique ID that specifies the destination chain for an order. However, the argument's value is not checked against the `trustedRemoteLookup` or `spokes` mappings to ensure it is a valid value.

Impact

The two order-placing functions transfer in some amount of funds, and these funds would be permanently locked in the contract.

Recommendations

Ensure the `trustedRemoteLookup[contra_cid]` and/or `spokes[contra_cid]` values are nonzero.

Remediation

This issue has been acknowledged by Tristero Incorporated, and a fix was implemented in commit 587cd946.

3.5 Lack of maker order-cancellation mechanism

- **Target**: Spoke
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

When a maker order is placed to provide liquidity to the protocol, a `MARGIN_BPS/MAXBPS` amount of collateral is transferred into the protocol.

If the maker order successfully funds upon matching with another taker order, the maker order remains in the maker order's linked list and is immediately ready to match with a new taker order. The collateral remains in the protocol.

If the maker order fails to fund, the collateral is seized and transferred to the DAO address, and the order is removed from the maker order's linked list.

However, there is no mechanism for the maker order placer to cancel their order and retrieve their collateral. They are forever forced to fund orders or allow their collateral to be seized.

Impact

Collateral is essentially seized as soon as a user places a maker order.

Recommendations

Provide a mechanism for the maker order placer to cancel their order. Consider using a timelock of epochs before the maker order is cancelled after the user requests cancellation, in which taker orders can still match the maker order pending cancellation.

Remediation

Tristero Incorporated noted that this behavior is intended.

3.6 Missing result check in `transferFrom` helper

- **Target**: Spoke
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

Hourglass uses the following functions to transfer ERC-20 tokens into and out of the contract:

```
//TransferFunctions
function transferFrom (address tkn, address from, uint amt)
    internal returns (bool s)
{ (s,) = tkn.call(abi.encodeWithSelector(IERC20_.transferFrom.selector,
    from, address(this), amt)); }

function transfer (address tkn, address to, uint amt) internal
{tkn.call(abi.encodeWithSelector(IERC20_.transfer.selector, to, amt));}
```

It is worth noting that the `transfer` helper function does not check the return value, so it could silently fail. But in practical, normal conditions, this should never happen in Hourglass.

More importantly, the return value is not checked in the `transferFrom` helper function.

Some ERC-20 tokens return specific values (such as `false`) instead of reverting upon transfer failure. Examples include:

- LDO
- VEN
- cETH
- HT

Impact

A user may be tricked into making a trade between two legitimate tokens where the transfer from the attacker to the victim does not successfully occur, but the transfer from the victim to the attacker does occur; that is, a malicious user could set up a trade that looks very favorable and trick a user into fulfilling the order without ever

transferring the source token.

Recommendations

Use SafeERC20's `safeTransferFrom` function instead of `transferFrom`, and `safeTransfer` instead of `transfer`. Alternatively, ensure that the result of the `transferFrom` function call is checked.

Remediation

This issue has been acknowledged by Tristero Incorporated, and a fix was implemented in commit a4522d85.

3.7 Limited number of maker orders can be placed

- **Target**: Spoke
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

A limited number of maker orders are allowed to be placed:

```
uint constant max_orders=10;

// [...]

//1.2 -- placeMaker
function placeMaker(address sell_token, address buy_token, uint lz_cid,
    uint96 _quantity) public nonReentrant {
    // [...]

    //increment the maker count
    require(selected_pair.mkr_count < max_orders, "!makerStack");

    selected_pair.mkr_count++;

    // [...]
}
```

When a maker order is placed, it cannot be canceled. It is only ever deleted if the order fails to fund. A maximum of 10 (`max_orders`) maker orders may be active (i.e., placed and not deleted).

Impact

A malicious user could limit the amount of liquidity the protocol has by placing several small orders to fill all 10 slots. These orders must be fulfillable or else the maker orders will be canceled, but it denies the opportunity for other users to provide liquidity using maker orders.

Recommendations

Removing the limit would allow gas griefing, so we cannot recommend that. Similarly, canceling smaller orders in favor of larger orders would create a DOS opportunity, so

we cannot recommend that either.

Rather, we recommend allowing the minimum buy-in for maker orders to increase based on the previously placed orders.

Alternatively, consider algorithms that would allow only the best (i.e., closest in amounts) maker orders to be considered when matching a taker order to a maker order, in a way that would prevent gas griefing.

Remediation

This issue has been acknowledged by Tristero Incorporated, and a fix was implemented in commit 587cd946.

3.8 Off-by-one epochspan check

- **Target**: Spoke
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The epochspan variable does not define the minimum number of seconds between epochs but rather that number minus one; that is, to be able to send, one must wait epochspan+1 seconds from the previous epoch.

Impact

An implication of this is that tests are not as intuitive to write. For example, to skip ahead n epochs, one must warp using this:

```
vm.warp((spokeSrc.epochspan() + 1)*n);
```

Recommendations

Make the following code change:

```
//2.5
function canResolve(address sell_token, address buy_token, uint lz_cid)
    public view returns(bool) {
    Pair storage selected_pair=book[lz_cid][sell_token][buy_token];
    return (!selected_pair.isAwaiting && (block.timestamp-selected_
        pair.index.timestamp)>epochspan);
    return (!selected_pair.isAwaiting && (block.timestamp-selected_
        pair.index.timestamp)≥epochspan);
}

// [...]

require(!selected_pair.isAwaiting, "!await lz inbound msg");
require (block.timestamp - uint(selected_pair.index.timestamp) >
    epochspan, "!await timestamp");
require (block.timestamp - uint(selected_pair.index.timestamp) ≥
```

```
    epochspan, "!await timestamp");
require(address(this).balance ≥ 2*MINGAS, "!gasLimit send");
```

Remediation

This issue has been acknowledged by Tristero Incorporated, and a fix was implemented in commit f759a064.

3.9 Hardcoded minimum order size

- **Target**: Spoke
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

There is a minimum taker or maker order size of 1e4:

```
require(_quantity > 1e4, "!minOrder");
```

Also, note that users are allowed to use arbitrary tokens.

Impact

Since tokens have varying decimals, 1e4 may be a very small amount for a stablecoin, making denial-of-service attacks more practical. Likewise, it may be a very large amount of stablecoin, causing rounding errors to become significant.

Recommendations

Dynamically calculate the minimum order size based on the stablecoin token's decimals — and consider caching the return value of that external call to `ERC20.decimals`.

Remediation

This issue has been acknowledged by Tristero Incorporated, and a fix was implemented in commit 587cd946.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Consider making `lzc` immutable

Consider defining `lzc` — which is only set from the constructor — as an immutable variable:

```
uint16 public lzc;
uint16 public immutable lzc;
// [...]
//Constructor
constructor (address _lzEndpoint, uint16 _lzc)
    NonblockingLzApp(_lzEndpoint) {
    lzc = _lzc;
    dao_address=msg.sender;
}
```

Note that this discussion point does not have any security impact.

4.2 Lack of fees

Note that Hourglass does not have any fee structure. There are at least two implications:

- There is no way for the protocol to use fees to recover from transferring funds into the Spoke contract to be used as gas and LayerZero fees.
- There is no incentive to become a maker.

Tristero Incorporated provided the following response:

> We just want to get distribution.... We are content subsidizing user fees until we prove product market fit.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

5.1 Module: Multichain.sol

Function: `placeMaker(address sell_token, address buy_token, uint256 lz_cid, uint96 _quantity)`

This function allows a user to become a liquidity provider for the Hourglass protocol by supplying stablecoins on this chain that will be swapped for stablecoins on another chain. There can only be a maximum of 10 active maker orders for a given pair, which could be abused to limit the liquidity available (see Finding 3.7). There is also no way for a user to cancel their maker order (see Finding 3.5).

Inputs

- `sell_token`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: This is the token the user wishes to contribute the liquidity for.
- `buy_token`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: This is the token the user wishes to receive on the destination chain.
- `lz_cid`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: This is the LayerZero chain ID of the destination chain.
- `_quantity`
  - **Control**: The user must have approved this contract 1% of `_amount` of `sell_token`.
  - **Constraints**: Must be greater than 1e4.
  - **Impact**: One percent of this value of `sell_token` will be transferred to the contract as collateral, and `_quantity` of the tokens will be listed as liquidity.

Branches and code coverage (including function calls)

Intended branches

- The 1% collateral for the `sell_token` is transferred to the contract, and a new maker order is created to provide `_quantity` amount of liquidity for the selected pair.
  - ☐ Test coverage

Negative behavior

- The transfer of the collateral fails.
  - ☐ Negative test
- The amount is less than 1e4.
  - ☐ Negative test
- There are already `max_orders` makers for the selected pair.
  - ☐ Negative test

Function call analysis

- `placeMaker` → `transferFrom(sell_token, msg.sender, (_quantity*MARGIN_BPS ) / MAXBPS)`
  - **External/Internal?** Internal.
  - **Argument control?** `sell_token` and `_quantity`.
  - **Impact**: This will transfer 1% of `_quantity` of `sell_token` from the user to the contract.

Function: `placeTaker(address sell_token, address buy_token, uint256 lz_cid, uint96 _quantity)`

This function allows a user to place a new order to swap a stablecoin on the current chain with a stablecoin on a different chain. There can only be a maximum of 10 active taker orders for a given pair, although this calculation is incorrect (see Finding 3.3).

Inputs

- `sell_token`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: This is the token the user wishes to sell.
- `buy_token`
  - **Control**: Full control.
  - **Constraints**: N/A.

  - **Impact**: This is the token the user wishes to receive on the destination chain.
- `lz_cid`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: This is the LayerZero chain ID of the destination chain.
- `_quantity`
  - **Control**: The user must have approved this contract `_amount` of `sell_token`.
  - **Constraints**: Must be greater than 1e4.
  - **Impact**: This amount of `sell_token` will be transferred to the contract and be swapped when a matching order is found.

Branches and code coverage (including function calls)

Intended branches

- An order is placed and `amount` of `sell_token` is transferred from the sender to the contract.
  ☐ Test coverage

Negative behavior

- There are already `max_orders` for the selected token pair.
  ☐ Negative test
- The transfer of `sell_token` fails.
  ☐ Negative test
- The amount of tokens is less than 1e4.
  ☐ Negative test

Function call analysis

- `placeTaker` → `transferFrom(sell_token, msg.sender, _quantity)`
  - **External/Internal?** Internal.
  - **Argument control?** `sell_token` and `_quantity`.
  - **Impact**: This will transfer `_quantity` of `sell_token` from the user to the contract.

Function: `send(address sell_token, address buy_token, uint256 lz_cid)`

This function allows anyone to resolve the orders of a selected pair for the current epoch. This will trigger the core logic that matches up orders, generate the LayerZero payload to send to the destination chain, and update the current state of the pair so that it is ready for the next epoch. The selected pair must not already be waiting for a

response from the destination chain.

Inputs

- `sell_token`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Used to look up the pair to resolve.
- `buy_token`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Used to look up the pair to resolve.
- `lz_cid`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Used to look up the pair to resolve and to determine the destination for the LayerZero payload.

Branches and code coverage

Intended branches

- Check if the branch has test coverage.
  ☐ Test coverage
- Include function calls.
  ☐ Test coverage

Negative behavior

- The pair is already awaiting an inbound message.
  ☐ Negative test
- Not enough time has passed since the last resolve.
  ☐ Negative test
- The contract does not have enough balance to send the LayerZero message.
  ☐ Negative test

Function call analysis

- `send` → `resolve_epoch(sell_token, buy_token, lz_cid)`
  - **External/Internal?** Internal.
  - **Argument control?** `sell_token`, `buy_token`, and `lz_cid`.
  - **Impact**: This is where the main logic for matching the orders occurs.

- `resolve_epoch` → `send_orders(sell_token, buy_token, lz_cid, 0)`
  - **External/Internal?** Internal.
  - **Argument control?** `sell_token`, `buy_token`, and `lz_cid`.
  - **Impact**: This function calculates which orders need to be sent to the other spoke as well as pulls funds from makers and removes them if they default.
- `resolve_epoch` → `send_taker_sum(sell_token, buy_token, lz_cid)`
  - **External/Internal?** Internal.
  - **Argument control?** `sell_token`, `buy_token`, and `lz_cid`.
  - **Impact**: This function removes and refunds any taker orders that are too old, then calculates the total sum of the remaining taker order vaules.
- `resolve_epoch` → `_lzSend(uint16(lz_cid), abi.encode(newPayload), payable (this), address(0x0), adapterParams, address(this).balance)`
  - **External/Internal?** Internal.
  - **Argument control?** `lz_cid`.
  - **Impact**: This function sends the generated payload to the destination chain via LayerZero, including any orders that should be paid out, the current state of the pair on this chain, and the amount that makers defaulted this epoch.

Function: `_nonblockingLzReceive(uint16, byte[], uint64, byte[] _payload )`

This function is called when a payload is recieved from another spoke via a call to `send` and will pay out any orders, update the state of the selected pair, then resolve the current epoch and send back a message to the source chain if required. It is only by the `_lzEndpoint` specified in the constructor, and the message must have originated from a trusted remote sender.

Inputs

- `_payload`
  - **Control**: This payload is generated by the `send` method of the source spoke.
  - **Constraints**: N/A.
  - **Impact**: This will be used to payout any required orders and update the state of the selected pair.

Branches and code coverage

Intended branches

- The orders are paid out and the state updated.
  - ☐ Test coverage

- The orders are paid out, the state is updated, the current epoch is resolved, and a message is sent back to the source chain.
  - ☐ Test coverage

Negative behavior

- The caller is not the `lzEndpoint`.
  - ☐ Negative test
- The sender is not a trusted remote.
  - ☐ Negative test
- The selected pair requires a response message, but there is not enough balance in the contract to cover the gas.
  - ☐ Negative test

Function call analysis

- `_nonblockingLzReceive → payout_orders(sell_token, payload.orders, selected_pair.sums.maker_default_quantity)`
  - **External/Internal?** Internal.
  - **Argument control?** `sell_token`.
  - **Impact**: This will pay out any orders that were matched by the source chain. As this happens before the funds are pulled from the makers, it is possible that a payout transfer happens and then the maker defaults (see Finding 3.1).
- `_nonblockingLzReceive → roll_taker_orders(sell_token, buy_token, lz_cid, qd)`
  - **External/Internal?** Internal.
  - **Argument control?** `sell_token` and `buy_token`.
  - **Impact**: If a maker defaults, this function will update the list of orders to take into account the amount that was not funded. When updating the value of an order, it should also update the sums of the selected pair (see Finding 3.1).
- `_nonblockingLzReceive → resolve_epoch(sell_token, buy_token, lz_cid)`
  - **External/Internal?** Internal.
  - **Argument control?** `sell_token` and `buy_token`.
  - **Impact**: This will resolve the current epoch for the selected pair and send back a message to the source chain with any orders to be paid out as well as the updated state of the pair on this chain.

Function: `setDaoAddress(address new_address)`

This function sets the address that will receive the collateral if a maker defaults. It is only callable by the current owner.

Inputs

- `new_address`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: This will be the new DAO address.

Branches and code coverage

Intended branches

- The DAO address is updated.
  - ☐ Test coverage

Negative behavior

- The caller is not the owner.
  - ☐ Negative test

Function: `setspoke(address _contraspoke, uint16 contra_cid)`

This function allows a new trusted remote to be set for a remote chain, allowing it to send LayerZero messages to this spoke. It is only callable by the current owner.

Inputs

- `_contraspoke`
  - **Control**: Full control.
  - **Constraints**: Cannot be the ID of this chain.
  - **Impact**: This address will be added as a trusted remote.
- `contra_cid`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: This is the ID of the remote chain where the trusted remote address will originate from.

Branches and code coverage

Intended branches

- A new trusted remote is set for the specified chain.
  - ☐ Test coverage

Negative behavior

- The caller is not the owner.
  - ☐ Negative test
- The remote chain ID is the same as this chain's ID.
  - ☐ Negative test

Function: `transferFrom(address tkn, address from, uint256 amt)`

This function is a helper function to transfer tokens from a user to this contract. It returns a boolean to indicate if the call reverted or not, but it does not check the return value of the call itself (see Finding 3.6).

Inputs

- `tkn`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The token to be transferred.
- `from`
  - **Control**: Will be the address of the maker or taker.
  - **Constraints**: N/A.
  - **Impact**: This is where the tokens will be transferred from.
- `amt`
  - **Control**: This is determined by which taker orders were matched.
  - **Constraints**: N/A.
  - **Impact**: This amount of the tokens will be transferred.

Branches and code coverage

Intended branches

- The tokens are transferred from the user.
  - ☐ Test coverage

Negative behavior

- The transfer is reverted.
  - ☐ Negative test
- The transfer returns false to indicate a failure.

☐ Negative test

Function call analysis

- `transferFrom` → `tkn.call(abi.encodeWithSelector(IERC20_.transferFrom.selector, from, address(this), amt))`
  - **External/Internal?** External.
  - **Argument control?** `tkn`.
  - **Impact**: This will transfer `amt` of `tkn` from `from` to this contract.

Function: `transfer(address tkn, address to, uint256 amt)`

This function will transfer a specific amount of a token to a user from this contract. It does not check whether the transfer was successful, so if the balance of this contract is insufficient, then it will silently fail.

Inputs

- `tkn`
  - **Control**: This will be from the token pair that is being resolved.
  - **Constraints**: N/A.
  - **Impact**: This token will be sent to the user.
- `to`
  - **Control**: This will be the address of the maker or taker.
  - **Constraints**: N/A.
  - **Impact**: This address will receive the tokens.
- `amt`
  - **Control**: This will be the amount from an order.
  - **Constraints**: N/A.
  - **Impact**: The amount of tokens received.

Branches and code coverage

Intended branches

- The tokens are transferred to the user.
  ☐ Test coverage

Negative behavior

- The contract does not have enough token balance to cover the transfer.
  ☐ Negative test

Function call analysis

- `transfer` → `tkn.call(abi.encodeWithSelector(IERC20_.transfer.selector, to, amt))`
  - **External/Internal?** External.
  - **Argument control?** `tkn`.
  - **Impact**: This will transfer `amt` of `tkn` from this contract to `to`.

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Hourglass contracts, we discovered nine findings. One critical issue was found. Two were of high impact, three were of medium impact, one was of low impact, and the remaining findings were informational in nature. Tristero Incorporated acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.