Zellic

Prepared for
Yang Zheng
Levi Rybalov
Arkhai

Prepared by
Sunwoo Hwang
Chongyu Lv
Zellic

January 27, 2026

Alkahest
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Arkhai from January 15th to January 22nd, 2026. During this engagement, Zellic reviewed Alkahest's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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:

- Could an attacker manipulate the attestation format to bypass expected escrow conditions?
- Are there any race conditions that could be exploited during the escrow fulfillment process?
- How vulnerable are the escrow contracts to front-running attacks when releasing funds?
- Are there any scenarios where funds could become permanently locked in the escrow contracts?
- Is there sufficient access control for critical functions within the escrow contracts?
- Could a malicious arbiter implementation drain funds from associated escrow contracts?
- Are there any race conditions that could be exploited during the escrow fulfillment process?
- How vulnerable are the escrow contracts to front-running attacks when releasing funds?

1.3. 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.

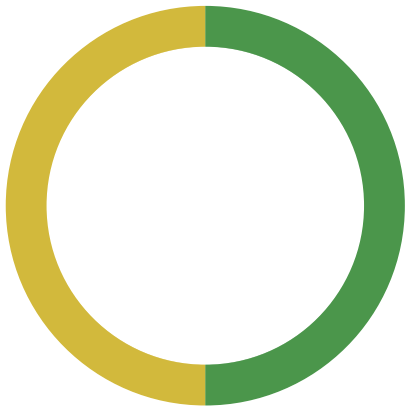
1.4. Results

During our assessment on the scoped Alkahest contracts, we discovered four findings. No critical issues were found. Two findings were of medium impact and two were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Arkhai in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	2
🟩 Low	2
⬜ Informational	0



2. Introduction

2.1. About Alkahest

Arkhai contributed the following description of Alkahest:

> Alkahest is a library and ecosystem of contracts for arbitrated peer-to-peer escrow. It contains three main types of contracts:
>
> - Escrow contracts conditionally guarantee an on-chain action (usually a token transfer).
> - Arbiter contracts represent the conditions on which escrows are released, and can be composed via AllArbiter and AnyArbiter.
> - Obligation contracts produce EAS attestations that represent the fulfillments to escrows, and which are checked by arbiter contracts. Escrow contracts are also obligation contracts.

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 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, upgradability 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:

Alkahest Contracts

Type	Solidity
Platform	EVM-compatible

Target	alkahest
Repository	https://github.com/arkhai-io/alkahest ↗
Version	`b73a7f30d4cbecf44f2dcaccc276efc4276b38d2`

Programs	`arbiters/attestation-properties/*.sol`
	`arbiters/confirmation/*.sol`
	`arbiters/logical/*.sol`
	`arbiters/ERC8004Arbiter.sol`
	`arbiters/IntrinsicsArbiter.sol`
	`arbiters/IntrinsicsArbiter2.sol`
	`arbiters/TrivialArbiter.sol`
	`arbiters/TrustedOracleArbiter.sol`
	`obligations/escrow/*.sol`
	`obligations/StringObligation.sol`
	`ArbiterUtils.sol`
	`BaseAttester.sol`
	`BaseEscrowObligation.sol`
	`BaseEscrowObligationTierable.sol`
	`BaseObligation.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of one person-week. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Sunwoo Hwang
Engineer
sunwoo@zellic.io

Chongyu Lv
Engineer
chongyu@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 15, 2026	Kick-off call
January 15, 2026	Start of primary review period
January 22, 2026	End of primary review period

3. Detailed Findings

3.1. Use of `transfer` / `transferFrom` can lead to denial of service and failures

Target	ERC20EscrowObligation, TokenBundleEscrowObligation		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `transfer` / `transferFrom` functions are used in nontierable ERC20EscrowObligation/TokenBundleEscrowObligation contracts to transfer ERC-20 tokens. The code implementation of `ERC20EscrowObligation._lockEscrow` is as follows:

```
// Transfer tokens into escrow
    function _lockEscrow(bytes memory data, address from) internal override {
        ObligationData memory decoded = abi.decode(data, (ObligationData));

        // Check balance before transfer
        uint256 balanceBefore
    = IERC20(decoded.token).balanceOf(address(this));

        bool success;
        try
            IERC20(decoded.token).transferFrom(
                from,
                address(this),
                decoded.amount
            )
        returns (bool result) {
            success = result;
        } catch {
            success = false;
        }

        // Check balance after transfer
        uint256 balanceAfter = IERC20(decoded.token).balanceOf(address(this));

        // Verify the actual amount transferred
        if (!success || balanceAfter < balanceBefore + decoded.amount) {
            revert ERC20TransferFailed(
                decoded.token,
                from,
                address(this),
                decoded.amount
            );
        }
    }
```

However, nonstandard tokens like USDT cannot be used properly in this situation. The reason is that the current code implementation (as of the time of writing) relies on `transfer` / `transferFrom` returning a `bool` and uses `try`/`catch` to determine success.

However, some ERC-20 tokens do not follow the interface convention of returning a `bool` (USDT is one of the most common examples). If the code receives the return value using `returns (bool)`, and the target token contract (USDT) does not return a value, this will cause ABI decoding to fail, it will be caught by the try-catch block, and `success` will be set to `false`, ultimately leading to `revert ERC20TransferFailed`.

Impact

When users use ERC-20 tokens such as USDT (which have nonstandard return values), `_lockEscrow` may trigger `revert ERC20TransferFailed`, causing related business processes to fail. Specific impacts include the following:

1. An inability to use tokens such as USDT to create escrows (functionality unavailable, resulting in a denial-of-service attack)
2. Upper-level paths relying on lockup (such as creation/fulfillment processes) failing due to unexpected reverts, reducing the protocol's composability and the range of available tokens

Recommendations

Consider using OpenZeppelin's `safeTransfer` / `safeTransferFrom` ↗ functions that handle the return-value check as well as non–standard-compliant tokens.

```
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {SafeERC20}
    from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

using SafeERC20 for IERC20;

uint256 balanceBefore = IERC20(decoded.token).balanceOf(address(this));
IERC20(decoded.token).safeTransferFrom(from, address(this), decoded.amount);
uint256 balanceAfter = IERC20(decoded.token).balanceOf(address(this));

if (balanceAfter < balanceBefore + decoded.amount) {
    revert ERC20TransferFailed(decoded.token, from, address(this),
    decoded.amount);
}
```

Remediation

PR #39

3.2. NativeToken transfers allow overpayments that are nonrefundable, causing the overpaid ETH to remain permanently stuck in the contract

Target	NativeTokenEscrowObligation, TokenBundleEscrowObligation		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Both `NativeTokenEscrowObligation._lockEscrow` and `TokenBundleEscrowObligation._lockEscrow` support ETH transfers (tierable and nontierable).

```
// `NativeTokenEscrowObligation._lockEscrow`
// Lock native tokens into escrow
    function _lockEscrow(
        bytes memory data,
        address /* from */
    ) internal override {
        ObligationData memory decoded = abi.decode(data, (ObligationData));

        if (msg.value < decoded.amount) {
            revert InsufficientPayment(decoded.amount, msg.value);
        }
    }

// `TokenBundleEscrowObligation._lockEscrow`
// Transfer tokens into escrow
    function _lockEscrow(bytes memory data, address from) internal override {
        ObligationData memory decoded = abi.decode(data, (ObligationData));
        validateArrayLengths(decoded);

        // Handle native tokens
        if (decoded.nativeAmount > 0) {
            if (msg.value < decoded.nativeAmount) {
                revert InsufficientPayment(decoded.nativeAmount, msg.value);
            }
        }

        // Handle token bundle
        transferInTokenBundle(decoded, from);
    }
```

Both of the above `_lockEscrow` paths accept declared amounts as `msg.value >= decoded.nativeAmount` but do not record the actual amount deposited for each escrow account, nor do they refund any excess. Therefore, any overpayment will permanently increase the contract balance.

Impact

Overpayment will remain permanently in the contract, and the code implementation lacks a withdrawal mechanism to retrieve excess ETH.

Here is a proof of concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.26;

import {Test} from "forge-std/Test.sol";

import {EAS} from "@eas/EAS.sol";
import {IEAS, Attestation} from "@eas/IEAS.sol";
import {SchemaRegistry} from "@eas/SchemaRegistry.sol";
import {ISchemaRegistry} from "@eas/ISchemaRegistry.sol";

import {IArbiter} from "@src/IArbiter.sol";
import {StringObligation} from "@src/obligations/StringObligation.sol";
import {NativeTokenEscrowObligation} from "@src/obligations/escrow/non-
    tierable/NativeTokenEscrowObligation.sol";

contract TrueArbiter is IArbiter {
    function checkObligation(Attestation memory, bytes memory, bytes32)
    external pure returns (bool) {
        return true;
    }
}

contract NativeTokenEscrowObligation_OverpaymentStuck_POC is Test {
    function _deployEAS() internal returns (IEAS eas,
    ISchemaRegistry registry) {
        SchemaRegistry schemaRegistry = new SchemaRegistry();
        EAS easImpl = new EAS(schemaRegistry);
        return (IEAS(address(easImpl)),
    ISchemaRegistry(address(schemaRegistry)));
    }

    function test_POC_Overpayment_Stuck() public {
        (IEAS eas, ISchemaRegistry schemaRegistry) = _deployEAS();
        NativeTokenEscrowObligation escrow
    = new NativeTokenEscrowObligation(eas, schemaRegistry);
        StringObligation stringObligation = new StringObligation(eas,
    schemaRegistry);
        TrueArbiter arbiter = new TrueArbiter();

        address alice = makeAddr("alice");
        address seller = makeAddr("seller");
        vm.deal(alice, 10 ether);
        vm.deal(seller, 1 ether);

        uint256 amount = 1 ether;
        uint256 overpay = 2 ether;

        NativeTokenEscrowObligation.ObligationData memory data
    = NativeTokenEscrowObligation.ObligationData({
            arbiter: address(arbiter), demand: abi.encode("demand"), amount:
    amount
        });

        // Alice accidentally overpays when creating the escrow.
        vm.prank(alice);
        bytes32 escrowUid = escrow.doObligation{value: overpay}(data,
    uint64(block.timestamp + 1 days));

        // Fulfillment references escrowUid (non-tierable requirement).
        vm.prank(seller);
        bytes32 fulfillmentUid =

    stringObligation.doObligation(StringObligation.ObligationData({item:
    "fulfillment"}), escrowUid);

        uint256 sellerBefore = seller.balance;

        vm.prank(seller);
        bool ok = escrow.collectEscrow(escrowUid, fulfillmentUid);
        assertTrue(ok, "collect ok");

        // Seller receives only `amount`, excess remains stuck in the escrow
    contract.
        assertEq(seller.balance, sellerBefore + amount, "seller paid amount");
        assertEq(address(escrow).balance, overpay - amount, "excess stuck");
    // poc here

        Attestation memory escrowAtt = eas.getAttestation(escrowUid);
        assertTrue(escrowAtt.revocationTime != 0, "escrow revoked");
    }
```

```
}
```

Recommendations

Consider changing the code implementation as follows:

```
// Handle native tokens
if (msg.value != decoded.amount) {
    revert InsufficientPayment(decoded.nativeAmount, msg.value);
}
```

Remediation

PR #40 ↗

3.3. Missing balance-difference check in `TokenBundleEscrowObligation.transferInTokenBundle`

Target	TokenBundleEscrowObligation		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

There is a balance-difference check in `ERC20EscrowObligation._lockEscrow`:

```
// Transfer tokens into escrow
    function _lockEscrow(bytes memory data, address from) internal override {
        ObligationData memory decoded = abi.decode(data, (ObligationData));

        // Check balance before transfer
        uint256 balanceBefore
    = IERC20(decoded.token).balanceOf(address(this));

        bool success;
        try
            IERC20(decoded.token).transferFrom(
                from,
                address(this),
                decoded.amount
            )
        returns (bool result) {
            success = result;
        } catch {
            success = false;
        }

        // Check balance after transfer
        uint256 balanceAfter = IERC20(decoded.token).balanceOf(address(this));

        // Verify the actual amount transferred
        if (!success || balanceAfter < balanceBefore + decoded.amount) {
            revert ERC20TransferFailed(
                decoded.token,
                from,
                address(this),
                decoded.amount
            );
        }
    }
```

However, the `TokenBundleEscrowObligation._lockEscrow` function does not implement a similar mechanism; `TokenBundleEscrowObligation._lockEscrow` calls the `transferInTokenBundle` function, as shown in the following code.

```
function transferInTokenBundle(
        ObligationData memory data,
        address from
    ) internal {
        // Transfer ERC20s
        for (uint i = 0; i < data.erc20Tokens.length; i++) {
            bool success;
            try
                IERC20(data.erc20Tokens[i]).transferFrom(
                    from,
                    address(this),
                    data.erc20Amounts[i]
                )
            returns (bool result) {
                success = result;
            } catch {
                success = false;
            }

            if (!success) {
                revert ERC20TransferFailed(
                    data.erc20Tokens[i],
                    from,
                    address(this),
                    data.erc20Amounts[i]
                );
            }
        }

        // Transfer ERC721s
        for (uint i = 0; i < data.erc721Tokens.length; i++) {
            // Check ownership before transfer
            address ownerBefore = IERC721(data.erc721Tokens[i]).ownerOf(
                data.erc721TokenIds[i]
            );
            if (ownerBefore != from) {
                revert ERC721TransferFailed(
                    data.erc721Tokens[i],
                    from,
                    address(this),
                    data.erc721TokenIds[i]
                );
            }

            try
                IERC721(data.erc721Tokens[i]).transferFrom(
                    from,
                    address(this),
                    data.erc721TokenIds[i]
                )
            {
                // Transfer succeeded
            } catch {
                revert ERC721TransferFailed(
                    data.erc721Tokens[i],
                    from,
                    address(this),
                    data.erc721TokenIds[i]
                );
            }

            // Check ownership after transfer
            address ownerAfter = IERC721(data.erc721Tokens[i]).ownerOf(
                data.erc721TokenIds[i]
            );
            if (ownerAfter != address(this)) {
                revert ERC721TransferFailed(
                    data.erc721Tokens[i],
                    from,
                    address(this),
                    data.erc721TokenIds[i]
                );
            }
        }

        // Transfer ERC1155s
        for (uint i = 0; i < data.erc1155Tokens.length; i++) {
            // Check balance before transfer
            uint256 balanceBefore = IERC1155(data.erc1155Tokens[i]).balanceOf(
                address(this),
                data.erc1155TokenIds[i]
            );

            try
```

```
                IERC1155(data.erc1155Tokens[i]).safeTransferFrom(
                    from,
                    address(this),
                    data.erc1155TokenIds[i],
                    data.erc1155Amounts[i],
                    ""
                )
            {
                // Transfer succeeded
            } catch {
                revert ERC1155TransferFailed(
                    data.erc1155Tokens[i],
                    from,
                    address(this),
                    data.erc1155TokenIds[i],
                    data.erc1155Amounts[i]
                );
            }

            // Check balance after transfer
            uint256 balanceAfter = IERC1155(data.erc1155Tokens[i]).balanceOf(
                address(this),
                data.erc1155TokenIds[i]
            );

            // Verify the actual amount transferred
            if (balanceAfter < balanceBefore + data.erc1155Amounts[i]) {
                revert ERC1155TransferFailed(
                    data.erc1155Tokens[i],
                    from,
                    address(this),
                    data.erc1155TokenIds[i],
                    data.erc1155Amounts[i]
                );
            }
        }
    }
```

Impact

Both ERC20EscrowObligation and TokenBundleEscrowObligation involve transferring ERC-20 tokens; therefore, a unified balance-difference check should be implemented. Although fee-on-transfer tokens are not supported by the protocol, similar functionalities should be implemented consistently during code implementation to avoid introducing discrepancies.

Recommendations

Consider adding a balance-difference check to TokenBundleEscrowObligation to align it with ERC20EscrowObligation.

Remediation

PR #39 ↗

3.4. TokenBundleEscrowObligation continues after failed transfers, permanently locking escrowed assets

Target	TokenBundleEscrowObligation		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

`TokenBundleEscrowObligation` (tierable and non-tierable) uses a continue execution + emit event strategy during escrow release. If a transfer of native/ERC20/ERC721/ERC1155 assets fails, the function does not revert, so the overall `collectEscrow`/`reclaimExpired` call can still succeed.

For example, in `_releaseEscrow`, native token transfers emit an event on failure instead of reverting, and the `transferOutTokenBundle` also continues on individual failures:

```
// Transfer tokens to fulfiller
    function _releaseEscrow(
        bytes memory escrowData,
        address to,
        bytes32 /* fulfillmentUid */
    ) internal override returns (bytes memory) {
        ObligationData memory decoded = abi.decode(
            escrowData,
            (ObligationData)
        );

        // Transfer native tokens - continue on failure
        if (decoded.nativeAmount > 0) {
            (bool success, ) = payable(to).call{value:  decoded.nativeAmount}(
                ""
            );
            if (!success) {
                emit NativeTokenTransferFailed(to, decoded.nativeAmount);
            }
        }

    // Transfer token bundle - continues on individual failures
    transferOutTokenBundle(decoded, to);
    return ""; // Token escrows don't return anything
}
```

Because `BaseEscrowObligation.collectEscrowRaw` / `reclaimExpired` revokes the escrow attestation before releasing assets. If `_releaseEscrow` does not revert, the transaction succeeds and the escrow is treated as settled, but any failed asset transfer leaves the asset permanently locked in the escrow contract.

Impact

Escrows can be marked as successfully collected/reclaimed while one or more assets are permanently stuck in the escrow contract.

Recommendations

Revert the entire `collectEscrow` / `reclaimExpired` transaction if any asset transfer fails during release. This ensures the attestation revocation and any other state changes are rolled back along with the failed transfer, preventing assets from becoming permanently stuck in the escrow contract.

Remediation

`PR #41` ↗

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 adding a reentrant lock

The code implementation as of the time of writing does not contain any feasible call chain to implement a reentrancy attack, making it impossible to exploit a reentrancy vulnerability to repeatedly release the same escrowed asset because the escrowed asset has already been revoked before `_releaseEscrow` is called. However, to prevent potential future code changes, adding a reentrancy lock could be considered as a precaution.

We believe the functions `collectEscrowRaw` and `reclaimExpired` could benefit from adding a reentrancy lock because they call the functions `_releaseEscrow` and `_returnEscrow`, both of which use ETH to send data. Adding a reentrancy lock to `BaseObligation._doObligationForRaw` could also be considered. This is for informational purposes only, as we have not found any reentrancy vulnerabilities in the scoped code.

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: BaseEscrowObligation

Function: `reclaimExpired(byte[32] uid)`

This function allows the caller to trigger the reclaim process after the specified escrow ticket EAS attestation has expired.

Inputs

- `uid`
    - **Control**: Full control.
    - **Constraints**: The attestation UID must exist.
    - **Impact**: The UID of the reclaim attestation to retrieve.

Branches and code coverage

Intended branches

- ☑ Expired fund collection should succeed.
- ☑ The buyer should have received tokens back.
- ☑ Escrow should have zero tokens left.

Negative behavior

- ☑ Revert the transaction if an attempt is made to collect before expiration.

5.2. Module: ERC20EscrowObligation

Function: `collectEscrow(byte[32] escrow, byte[32] fulfillment)`

This function is used to settle an ERC-20 escrow order.

Inputs

- `escrow`

    - **Control**: Full control.
    - **Constraints**: The attestation UID must exist.
    - **Impact**: The UID of the escrow attestation to retrieve.
- `fulfillment`
    - **Control**: Full control.
    - **Constraints**: The attestation UID must exist.
    - **Impact**: The UID of the fulfillment attestation to retrieve.

Branches and code coverage

Intended branches

- ☑ Payment collection should succeed.
- ☑ The seller should have received tokens.
- ☑ Escrow should have zero tokens left.

Negative behavior

- ☑ Revert if fulfillment is invalid.

Function: `doObligation(ObligationData data, uint64 expirationTime)`

This function is used to create an ERC-20 escrow order and cast the corresponding escrow attestation on EAS.

Inputs

- `data`
    - **Control**: Full control.
    - **Constraints**: It must be decoded into `ObligationData(arbiter, demand, token, amount)`.
    - **Impact**: The ABI-encoded byte string of the `ObligationData` for this escrow order.
- `expirationTime`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: The time when the attestation expires (Unix timestamp).

Branches and code coverage

Intended branches

- ☑ The attestation should be successfully created.
- ☑ The attestation schema should match.
- ☑ Escrow should hold tokens.
- ☑ The buyer should have sent tokens.
- ☐ Nonstandard ERC-20 tokens such as USDT should be able to be used normally.

5.3. Module: ERC721EscrowObligation

Function: `collectEscrow(byte[32] escrow, byte[32] fulfillment)`

This function is used to settle an ERC-721 escrow order.

Inputs

- `escrow`
  - **Control**: Full control.
  - **Constraints**: The attestation UID must exist.
  - **Impact**: The UID of the escrow attestation to retrieve.
- `fulfillment`
  - **Control**: Full control.
  - **Constraints**: The attestation UID must exist.
  - **Impact**: The UID of the fulfillment attestation to retrieve.

Branches and code coverage

Intended branches

- ☑ Payment collection should succeed.
- ☑ The seller should have received tokens.
- ☑ Escrow should have zero tokens left.

Negative behavior

- ☑ Revert if fulfillment is invalid.

Function: `doObligation(ObligationData data, uint64 expirationTime)`

This function is used to create an ERC-721 escrow order and cast the corresponding escrow attestation on EAS.

Inputs

- `data`
    - **Control**: Full control.
    - **Constraints**: It must be decoded into `ObligationData(arbiter, demand, token, tokenId)`.
    - **Impact**: The ABI-encoded byte string of the `ObligationData` for this escrow order.
- `expirationTime`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: The time when the attestation expires (Unix timestamp).

Branches and code coverage

- ✓ The attestation should be successfully created.
- ✓ The attestation schema should match.
- ✓ Escrow should hold tokens.
- ✓ The buyer should have sent tokens.

5.4. Module: ERC1155EscrowObligation

Function: `collectEscrowRaw(byte[32] _escrow, byte[32] _fulfillment)`

This function is used to settle an ERC-1155 escrow order.

Inputs

- `_escrow`
    - **Control**: Full control.
    - **Constraints**: The attestation UID must exist.
    - **Impact**: The UID of the escrow attestation to retrieve.
- `_fulfillment`
    - **Control**: Full control.
    - **Constraints**: The attestation UID must exist.
    - **Impact**: The UID of the fulfillment attestation to retrieve.

Branches and code coverage

Intended branches

- ☑ Payment collection should succeed.
- ☑ The seller should have received tokens.
- ☑ Escrow should have zero tokens left.

Negative behavior

- ☑ Revert if fulfillment is invalid.

Function: `doObligation(ObligationData data, uint64 expirationTime)`

This function is used to create an ERC-1155 escrow order and cast the corresponding escrow attestation on EAS.

Inputs

- `data`
  - **Control**: Full control.
  - **Constraints**: It must be decoded into `ObligationData(arbiter, demand, token, tokenId, amount)`.
  - **Impact**: The ABI-encoded byte string of the `ObligationData` for this escrow order.
- `expirationTime`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: The time when the attestation expires (Unix timestamp).

Branches and code coverage

Intended branches

- ☑ The attestation should be successfully created.
- ☑ The attestation schema should match.
- ☑ Escrow should hold tokens.
- ☑ The buyer should have sent tokens.

5.5. Module: NativeTokenEscrowObligation

Function: `collectEscrowRaw(byte[32] _escrow, byte[32] _fulfillment)`

This function is used to settle a native token escrow order.

Inputs

- `_escrow`
    - **Control**: Full control.
    - **Constraints**: The attestation UID must exist.
    - **Impact**: The UID of the escrow attestation to retrieve.
- `_fulfillment`
    - **Control**: Full control.
    - **Constraints**: The attestation UID must exist.
    - **Impact**: The UID of the fulfillment attestation to retrieve.

Branches and code coverage

Intended branches

- ☑ Payment collection should succeed.
- ☑ The seller should have received tokens.
- ☑ Escrow should have zero tokens left.

Negative behavior

- ☑ Revert if fulfillment is invalid.

Function: `doObligation(ObligationData data, uint64 expirationTime)`

This function is used to create a native token escrow order and cast the corresponding escrow attestation on EAS.

Inputs

- `data`
    - **Control**: Full control.
    - **Constraints**: It must be decoded into `ObligationData(arbiter, demand, amount)`.

    - **Impact**: The ABI-encoded byte string of the `ObligationData` for this escrow order.
- `expirationTime`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: The time when the attestation expires (Unix timestamp).

Branches and code coverage

Intended branches

- ☑ The attestation should be successfully created.
- ☑ The attestation schema should match.
- ☑ Escrow should hold tokens.
- ☑ The buyer should have sent tokens.

5.6. Module: TokenBundleEscrowObligation

Function: `collectEscrow(byte[32] escrow, byte[32] fulfillment)`

This function is used to settle a token bundle escrow order.

Inputs

- `escrow`
    - **Control**: Full control.
    - **Constraints**: The attestation UID must exist.
    - **Impact**: The UID of the escrow attestation to retrieve.
- `fulfillment`
    - **Control**: Full control.
    - **Constraints**: The attestation UID must exist.
    - **Impact**: The UID of the fulfillment attestation to retrieve.

Branches and code coverage

Intended branches

- ☑ Payment collection should succeed.
- ☑ The seller should have received tokens.
- ☑ Escrow should have zero tokens left.

Negative behavior

- ☑ Revert if fulfillment is invalid.

Function: `_checkTokenArrays(ObligationData payment, ObligationData demand)`

This function is used to create a token bundle escrow order and cast the corresponding escrow attestation on EAS.

Inputs

- `data`
  - **Control**: Full control.
  - **Constraints**: It must be decoded into `ObligationData(arbiter, demand, nativeAmount, erc20Tokens, erc20Amounts, erc721Tokens, erc721TokenIds, erc1155Tokens, erc1155TokenIds, erc1155Amounts)`.
  - **Impact**: The ABI-encoded byte string of the `ObligationData` for this escrow order.
- `expirationTime`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: The time when the attestation expires (Unix timestamp).

Branches and code coverage

Intended branches

- ☑ The attestation should be successfully created.
- ☑ The attestation schema should match.
- ☑ Escrow should hold tokens.
- ☑ The buyer should have sent tokens.
- ☐ Nonstandard ERC-20 tokens such as USDT should be able to be used normally.

6. Assessment Results

During our assessment on the scoped Alkahest contracts, we discovered four findings. No critical issues were found. Two findings were of medium impact and two were of low impact.

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.