Bando

SMART CONTRACTS REVIEW

zokyo

November 25th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING BANDO COOL S.A. DE C.V.

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 3 Critical issues: 3 resolved = 0 points deducted
- 1 High issue: 1 resolved = 0 points deducted
- 8 Medium issues: 7 resolved and 1 acknowledged = - 3 points deducted
- 1 Low issue: 1 resolved = 0 points deducted
- 6 Informational issues: 6 resolved = 0 points deducted

Thus, 100 - 3 = 97

TECHNICAL SUMMARY

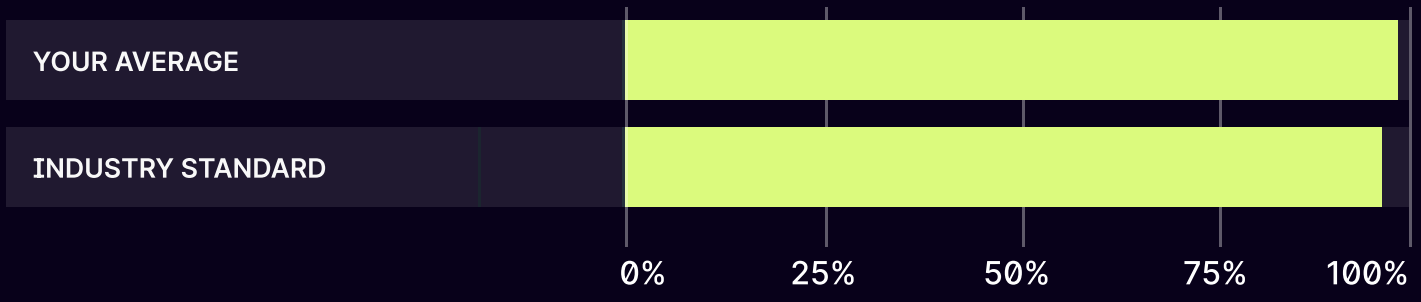
This document outlines the overall security of the BANDO COOL S.A. DE C.V. smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the BANDO COOL S.A. DE C.V. smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 3 critical issues found during the review. (See Complete Analysis)

Testable Code



95,35% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the BANDO COOL S.A. DE C.V. team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

Auditing Strategy and Techniques Applied 5

Executive Summary 7

Structure and Organization of the Document 8

Complete Analysis 9

Code Coverage and Test Results for all files written by Zokyo Security 29

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the BANDO COOL S.A. DE C.V. repository:
Repo: https://github.com/bandohq/evm-fulfillment-protocol
Initial commit: 894496848f47be70d1870486235dff386d953aca
Last fixes: https://github.com/bandohq/evm-fulfillment-protocol/pull/82

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./BandoERC20FulfillableV1.sol
- ./BandoFulfillmentManagerV1.sol
- ./IBandoERC20Fulfillable.sol
- ./libraries/FulfillmentRequestLib.sol
- ./periphery/registry/IFulfillableRegistry.sol
- ./periphery/registry/ERC20TokenRegistry.sol
- ./periphery/registry/FulfillableRegistry.sol
- ./periphery/registry/IERC20TokenRegistry.sol
- ./BandoRouterV1.sol
- ./BandoFulfillableV1.sol
- ./FulfillmentTypes.sol
- ./IBandoFulfillable.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of BANDO COOL S.A. DE C.V. smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Foundry testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contract/s by industry leaders.
03	Testing contract/s logic against common and uncommon attack vectors.
04	Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has performed a security audit of the provided codebase. The contracts submitted for auditing are well-crafted and organized. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

BANDO COOL S.A. DE C.V. protocol is about enabling an easy and seamless way for web3 users to spend via gift cards, airtime top-ups, bill payments, in-game items, digital goods, and any other fulfillable product or service. The protocol does this with a set of smart contracts that register and route user spend requests to the corresponding fulfiller to make the request whole.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the BANDO COOL S.A. DE C.V. team and the BANDO COOL S.A. DE C.V. team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Lack of an implementation for withdrawing beneficiaries balance	Critical	Resolved
2	Fund locked due to lack of an implementation for withdrawing service fees	Critical	Resolved
3	Loss of fulfiller funds and fees due to wrong comparison between depositsAmount and total_amount	Critical	Resolved
4	Lack of an implementation for taking service fees in ERC20 service requests	High	Resolved
5	There is no reason for requestERC20Service to be payable - can lead to fund loss	Medium	Resolved
6	Lack of updating _serviceCount variable	Medium	Resolved
7	High slippage between ETH based requests between request and fulfillment	Medium	Acknowledged
8	Fulfilments are based on variable service fee	Medium	Resolved
9	Users can't withdraw their own refunds themselves	Medium	Resolved
10	payer is not tied to msg.sender	Medium	Resolved
11	`disableInitializers` not used to prevent uninitialized contracts	Medium	Resolved
12	The removeServiceAddress() function fails with Overflow error	Medium	Resolved
13	Lack of emitting events	Low	Resolved
14	Floating pragma	Informational	Resolved

#	Title	Risk	Status
15	Accidentally setting fulfiler address to 0 has effect of disabling service	Informational	Resolved
16	Pre and post balance checks on msg.sender is not needed	Informational	Resolved
17	Consider using custom errors instead of strings	Informational	Resolved
18	Unnecessary check	Informational	Resolved
19	Duplicated nonReentrant modifier calls	Informational	Resolved

CRITICAL-1 | RESOLVED

Lack of an implementation for withdrawing beneficiaries balance

Location: BandoFulfillmentManagerV1.sol

The beneficiaryWithdraw() function in both BandoERC20FulfillableV1 and BandoFulfillableV1 contracts are only designed to be called by the manager contract but the BandoFulfillmentManagerV1 contract is missing an implementation for withdrawing the beneficiaries available balances.

```
function beneficiaryWithdraw(uint256 serviceID, address token) public
virtual nonReentrant {
        require(_manager == msg.sender, "Caller is not the manager");
        ...
}
```

Recommendation:

Add a function to withdraw beneficiaries available balances in the BandoFulfillmentManagerV1 contract.

Fund locked due to lack of an implementation for withdrawing service fees

Location: BandoRouterV1.sol

In the BandoRouterV1 contract, the requestService() function requires the caller to pay the sum of request.weiAmount and service.feeAmount.

```
(bool success, uint256 total_amount) =
request.weiAmount.tryAdd(service.feeAmount);
if (!success) {
    revert OverflowError();
}

if (msg.value != total_amount) {
    revert AmountMismatch();
}
```

But the function calls the _escrow.deposit() function with only request.weiAmount and service.feeAmount is locked in the contract.

```
IBandoFulfillable(_escrow).deposit{value: request.weiAmount}(serviceID,
request);
```

There is no implementation to withdraw this service fee in the contract and the fees are locked forever.

Recommendation:

Add a function to withdraw the service fees locked in the contract.

Loss of fulfiller funds and fees due to wrong comparison between depositsAmount and total_amount

Location: BandoERC20FulfillableV1.sol
Walkthrough:
Say a user will like to submit a request for a ERC20Service.

Initial states:

```
function getERC20DepositsFor(address token, address payer, uint256
serviceID) public view returns (uint256 amount) {
    amount = _erc20_deposits[serviceID][token][payer];
}
```

The initial deposit for the user is 0.

So when the user calls the router contract:

```
function requestERC20Service(
    uint256 serviceID,
    ERC20FulFillmentRequest memory request
) public payable whenNotPaused nonReentrant returns (bool)
```

The user transfers in request.tokenAmount, and thus deposits request.tokenAmount in the BandoERC20FulfillableV1.sol contract.

```
IERC20(request.token).safeTransferFrom(
     msg.sender,
    _erc20Escrow,
    request.tokenAmount
);
require(
    IERC20(request.token).balanceOf(msg.sender) <= pre_balance -
request.tokenAmount,
    "BandoRouterV1: ERC20 invalid transfer return"
);
IBandoERC20Fulfillable(_erc20Escrow).depositERC20(serviceID, request);
```

In the depoistERC20 function in the BandoERC20Fulfillable contract, amount is set as

```
uint256 amount = fulfillmentRequest.tokenAmount;
```

Since users current Deposit is 0

When setting their new deposit amount, their deposit is result, which is (0 + request.tokenAmount = request.tokenAmount)

```
(bool success, uint256 result) = amount.tryAdd(depositsAmount);
require(success, "Overflow while adding deposits");
setERC20DepositsFor(
    token,
    fulfillmentRequest.payer,
    serviceID,
    result
);
```

So the user has request.tokenAmount in their deposit balance for the service ID.

Also

```
ERC20FulFillmentRecord memory fulfillmentRecord = ERC20FulFillmentRecord({
    id: _fulfillmentIdCount,
    serviceRef: fulfillmentRequest.serviceRef,
    externalID: "",
    fulfiller: service.fulfiller,
    entryTime: block.timestamp,
    payer: fulfillmentRequest.payer,
    tokenAmount: fulfillmentRequest.tokenAmount,
    feeAmount: service.feeAmount,
    fiatAmount: fulfillmentRequest.fiatAmount,
    receiptURI: "",
    status: FulFillmentResultState.PENDING,
    token: fulfillmentRequest.token
});
```

```
_fulfillmentRecords[fulfillmentRecord.id] = fulfillmentRecord;
```

_fulfilmentRecords[fulfillment.id].tokenAmount = request.tokenAmount as seen above

Now when it's time to fulfill:

```
function registerFulfillment(uint256 serviceID, FulFillmentResult memory
fulfillment) public virtual nonReentrant returns (bool) {
…
uint depositsAmount = getERC20DepositsFor(
    token,
    _fulfillmentRecords[fulfillment.id].payer,
    serviceID
);
(bool ffsuccess, uint256 total_amount) =
_fulfillmentRecords[fulfillment.id].tokenAmount.tryAdd(
    service.feeAmount
);
require(ffsuccess, "Overflow while adding fulfillment amount and fee");
require(depositsAmount >= total_amount, "There is not enough balance to be
released");
```

So the check:

```
require(depositsAmount >= total_amount, "There is not enough balance to be
released");
```

Is actually checking:

```
require(request.tokenAmount >= request.tokenAmount + service.feeAmount,
"There is not enough balance to be released");
```

Which is always false when there is a service fee, causing the whole transaction to fail.

This same issue for requesting a service using native tokens.

Recommendation:

Service fees need to be properly accounted for when the user deposits and not when the fulfiller fulfills to avoid transaction failure after the fullfiler does their task.

HIGH-1 | RESOLVED

Lack of an implementation for taking service fees in ERC20 service requests

Location: BandoRouterV1.sol

The requestERC20Service() function is missing to take the service fee by transferring only request.tokenAmount from the caller even if the requestService() function has a check to make sure that msg.value is equal to the sum of request.weiAmount and service.feeAmount.

Recommendation:

Calculate the total amount of the request.tokenAmount and service.feeAmount and transfers the total amount from the caller.

MEDIUM-1 | RESOLVED

There is no reason for requestERC20Service to be payable - can lead to fund loss

Location: BandoRouterV1.sol

In the current implementation, the requestERC20Service() function is payable, which means that users can send ETH along with the transaction call, however the functions has no operations to do with ETH, meaning that if a user accidentally sends ETH along with their function call, the ETH will be trapped in the contract which leads to fund loss for the user.

Recommendation:

Remove payable keyword.

MEDIUM-2 | RESOLVED

Lack of updating _serviceCount variable

Location: FulfillableRegistry.sol
The addService() function is missing to update the _serviceCount variable.
Due to a lack of increasing the variable, the removeServiceAddress() function will always fail.

Recommendation:

Update the _serviceCount variable in the function.

MEDIUM-3 | ACKNOWLEDGED

High slippage between ETH based requests between request and fulfillment

ETH is a very volatile asset and there is a concern if an asset like ETH is compatible with what the project is trying to achieve. A high level overview of the protocol is that a user can pay in crypto on-chain for off chain services to be fulfilled. Consider this scenario, a user wants to buy a $100 gift card in eth, so they submit a request paying 11,235 GWEI (approximately $100 at the time of writing).

Scenario 1) Consider if the price of ETH rises by 10% from the time of the request and before the fulfiller fulfills the transaction. This will mean the user has now paid $110 worth of eth for a $100 gift card, and since there is no partial refund, the fulfiller profits and extra $10

Scenario 2) Consider if the price of ETH falls by 10% from the time of the request and before the fulfiller fulfills the transaction. This will mean that the user has paid $90 worth of eth for a $100 gift card. Unless the fulfiller wants to lose money, the transaction will not be executed and a cancellation will likely be forced.

Recommendation:

It is probably safer and more consistent to restrict this protocol to handle stable coins only. A suggestion would also to be to implement ERC-2612 Permit to extend functionality of your contract to allow gasless payments
https://eips.ethereum.org/EIPS/eip-2612

Client comment: Although it doesnt actually completely solve the issue. We introduced a "swap fee" for volatile assets to mitigate slippage and volatility. As we find it paramount for our business to support a wide range of tokens and assets to give them more utility. https://github.com/bandohq/evm-fulfillment-protocol/pull/76

MEDIUM-4 | RESOLVED

Fulfilments are based on variable service fee

In the current implementation of the protocol, when registering a fulfillment, the service fee is fetched from the registry contract.

```
Service memory service = _registryContract.getService(serviceID);
```

The problem is that during any time of the protocols operation the owner can change the service fee.

```
function updateServiceFeeAmount(uint256 serviceId, uint256 newFeeAmount)
external onlyOwner {
    require(_serviceRegistry[serviceId].fulfiller != address(0),
'FulfillableRegistry: Service does not exist');
    _serviceRegistry[serviceId].feeAmount = newFeeAmount;
}
```

The protocol is designed to handle many requests. If the service fee is changed between a user's request and the fulfillment, this can cause the fulfillment transaction to fail as the accounting is off and the user has not sent the fee amount that corresponds to the new fee. E.g if a user initially requested when the fees were 1 and a fee change to 2 occurs before the request is fulfilled, the fulfillment transaction will fail due to the user not having not sent enough funds to cover the new fee.

Recommendation:

Cache the fee that the user requested on at the time of the request, so even if there is a fee change before the user's request is fulfilled, they will still be charged based on the fee they submitted at the time of request.

Users can't withdraw their own refunds themselves

Location: BandoERC20FulfillableV1.sol, BandoFulfillableV1.sol

In the current implementation, if a user has a pending refund, they have to wait for the service fulfiler or the owner to process the refund.

```
function withdrawERC20Refund(uint256 serviceID, address token, address
refundee) public virtual nonReentrant returns (bool) {
```

```
    require(_manager == msg.sender, "Caller is not the manager");
    ...
}
```

This is an unusual pattern for a number of reasons

1. At periods of high network traffic, it will take long for a single entity to process refunds for all the users
2. Processing refunds costs gas for the fulfiler so the cost of processing a refund can cause financial loss as they are a) paying gas to fulfill the transaction b) paying gas again to submit the refund
3. If the fulfiler or owner ever goes offline, users refunds get stuck, violating the decentralized nature of the protocol
4. The users refund is already made available to them through the mapping:

If the refunds are already authorized, there is no harm with letting a user withdraw their own refunds

Recommendation:

If user refunds are already authorized, let them be free to withdraw it when they want. This will save gas for the protocol and time for the user, while mitigating risk of funds being unduly trapped

MEDIUM-6 | RESOLVED

payer is not tied to msg.sender

Bando fulfillment requests have the field “payer”. This field is user imputed and does not necessarily need to reflect the person making the transaction. This means a malicious user can create spam fake transactions on behalf of another user, which the fulfiler will eventually need to refund. At the minimum this leads to fulfiler loss as they have to spend gas submitting the fulfillment transactions and submitting refunds.

Out of scope thoughts: It is unclear how the protocol punishes spam, however if the fulfiler sees spam from a specific payer they might not be included to process their transactions. But if an attacker is impersonating a regular user, this penalty will be on a regular user.

Recommendation:

In the router contract ensure request.payer == msg.sender. If you want to allow 3rd parties to submit transactions on behalf of other users, you may consider implementing signed transactions to your protocol.

MEDIUM-7 | RESOLVED

`disableInitializers` not used to prevent uninitialized contracts

All the contracts are designed to be upgradeable.

In order to prevent leaving an implementation contract uninitialized it is recommended adding the _disableInitializers() function in the constructor to lock the contracts automatically when they
are deployed.

Recommendation:

Add `_disableInitializers()` to the constructor.

The removeServiceAddress() function fails with Overflow error

In FulfillableRegistry.sol,the removeServiceAddress() function fails with an Underflow error. This is because _serviceCount is never incremented when adding the service.

```
   function addService(uint256 serviceId, Service memory service) external
onlyManager returns (bool) {
       require(
           _serviceRegistry[serviceId].fulfiller == address(0),
           'FulfillableRegistry: Service already exists'
       );
       _serviceRegistry[serviceId] = service;
       // @audit Line below Added by auditor to test
       _serviceCount++;
       emit ServiceAdded(serviceId, service.fulfiller);
       return true;
   }
```

Recommendation:

Please consider incrementing the _serviceCount when adding a new service in the function such as the example given below

LOW-1 | RESOLVED

Lack of emitting events

Location: BandoERC20FulfillableV1.sol, BandoFulfillableV1.sol, BandoFulfillmentManagerV1.sol, BandoRouterV1.sol, FulfillableRegistry.sol

- setManager(), setRouter(), setFulfillableRegistry() functions in the BandoERC20FulfillableV1 and BandoFulfillableV1 contracts.
- setServiceRegistry(), setEscrow(), setERC20Escrow(), setService(), setServiceRef() functions in the BandoFulfillmentManagerV1 contract.
- setFulfillableRegistry(), setTokenRegistry(), setEscrow(), setERC20Escrow() functions in the BandoRouterV1 contract.
- setManager(), updateServiceBeneficiary(), updateServiceFeeAmount(), updateServiceFulfiller(), addFulfiller(), addServiceRef() functions in the FulfillableRegistry contract.

The above functions are missing events when key storages are updated.

Recommendation:

Add relative events in the functions.

INFORMATIONAL-1 | RESOLVED

Floating pragma

Location: All contracts

The contracts should not use floating pragma, e.g. (pragma solidity >=0.8.20 <0.9.0;), which allows a range of compiler versions. It is important to lock the pragma to prevent contracts from being accidentally deployed using a compiler with unfixed bugs.

Recommendation:
Fix the pragma version to a single version to ensure consistent behavior.

INFORMATIONAL-2 | RESOLVED

Accidentally setting fulfiler address to 0 has effect of disabling service

Location: FulfillableRegistry.sol

The FulfillableRegistry has lots of checks that revert if the fulfiller is the zero address.

There is a way to accidentally disable a service by setting the fulfiller to zero after the service fields have been set.

```
function updateServiceFulfiller(uint256 serviceId, address newFulfiller)
external onlyOwner {
    require(_serviceRegistry[serviceId].fulfiller != address(0),
'FulfillableRegistry: Service does not exist');
    _serviceRegistry[serviceId].fulfiller = newFulfiller;
}
```

There are no checks that the newFulfiller is not the 0 address.

Recommendation:
Add a check that verifies the newFulfiller address is not the 0 address.

Pre and post balance checks on msg.sender is not needed

Location: BandoRouterV1.sol

In the requestERC20Service() function, the additional safety checks on the pre and post balance of msg.sender are not needed, and a gas spending external calls.

```
require(pre_balance >= request.tokenAmount, "BandoRouterV1: Insufficient
balance");
    IERC20(request.token).safeTransferFrom(
    msg.sender,
    _erc20Escrow,
    request.tokenAmount
);
require(
    IERC20(request.token).balanceOf(msg.sender) <= pre_balance -
request.tokenAmount,
    "BandoRouterV1: ERC20 invalid transfer return"
```

```
);
```

The safeTransferFrom function already has built-in safety checks ensuring that the function will revert if the user doesn't have enough balance, this is also true for tokens that return a bool instead of reverting.

Recommendation:
Removing the pre and post balance checks allows gas to be saved by avoiding unnecessary computation and external calls.

There is another pattern you could use if you wanted to account for fee-on-transfer tokens, but these types of tokens don't seem compatible with the protocol and should not be whitelisted.

INFORMATIONAL-4 | RESOLVED

Consider using custom errors instead of strings

Location: BandofulfillableV1.sol

Using custom errors is more gas-efficient than reverting with strings because they reduce storage and execution costs by not storing string literals. Custom errors also improve code readability by providing more structured and specific failure conditions.

Recommendation:
Consider using custom errors to save gas.
(https://soliditylang.org/blog/2021/04/21/custom-errors/)

INFORMATIONAL-5 | RESOLVED

Unnecessary check

Location: BandoFulfillableV1.sol, BandoERC20FulfillableV1.sol

In the _authorizeRefund() function, the deposits variable is checked with both weiAmount and total_refunds.
However, the total_refunds is the sum of weiAmount and authorized_refunds and its value is always equal or greater than weiAmount.
So the comparison between “deposits” and “weiAmount” is not necessary because the “deposits” is compared with “total_refunds” again and it’s only passed when “deposits >= total_refunds”.
“deposits >= total_refunds” means that “deposits >= weiAmount”.

Recommendation:
Remove the comparison between “deposits” and “weiAmount”.

Duplicated nonReentrant modifier calls

Location: BandoFulfillmentManagerV1.sol, BandoRouterV1.sol

In the BandoFulfillmentManagerV1 contract, the registerFulfillment(), withdrawERC20Refund(), and registerERC20Fulfillment() functions have the nonReentrant modifiers and the functions make the external calls to the _escrow and _erc20_escrow contracts.
However, the external calls also have the nonReentrant modifiers so the entire transactions will have duplicated nonReentrant modifier calls.
The same for the requestERC20Service() and requestService() functions in the BandoRouterV1 contract.

Recommendation:
Remove nonReentrant modifiers in the functions of the BandoFulfillmentManagerV1 and BandoRouterV1 contracts.

	./BandoERC20FulfillableV1.sol **./BandoFulfillmentManagerV1.sol** **./IBandoERC20Fulfillable.sol** **./libraries/FulfillmentRequestLib.sol** **./periphery/registry/IFulfillableRegistry.sol** **./periphery/registry/ERC20TokenRegistry.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./periphery/registry/FulfillableRegistry.sol **./periphery/registry/IERC20TokenRegistry.sol** **./BandoRouterV1.sol** **./BandoFulfillableV1.sol** **./FulfillmentTypes.sol** **./IBandoFulfillable.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting BANDO COOL S.A. DE C.V. in verifying the correctness of their contract/s code, our team was responsible for writing integration tests using the Foundry testing framework.

The tests were based on the functionality of the code, as well as a review of the BANDO COOL S.A. DE C.V. contract/s requirements for details about issuance amounts and how the system handles these.

Tests Passed

✓ check setter functions
✓ check depositERC20
✓ check register fulfillment
✓ check withdrawRefund
✓ check beneficiary withdraw
✓ check router interactions
✓ check manager interactions
✓ check library functions

Failed Tests

1) The function removeServiceAddress() fails with an Underflow error. This is because _serviceCount is never incremented when adding the service. Please consider incrementing the _serviceCount when adding a new service.

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

	% Lines	% Statements	% Branches	% Funcs
contracts/BandoERC20FulfillableV1.sol	97.78% (88/90)	98.06% (101/103)	68.18% (30/44)	70.59% (12/17)
contracts/BandoFulfillableV1.sol	100.00% (80/80)	100.00% (94/94)	84.09% (37/44)	94.12% (16/17)
contracts/BandoFulfillmentManagerV1.sol	87.10% (27/31)	85.71% (30/35)	57.14% (16/28)	81.82% (9/11)
contracts/BandoRouterV1.sol	100.00% (26/26)	100.00% (27/27)	100.00% (12/12)	90.00% (9/10)
contracts/libraries/FulfillmentRequestLib.sol	82.61% (19/23)	84.62% (22/26)	55.56% (5/9)	100.00% (2/2)
contracts/periphery/registry/ERC20TokenRegistry.sol	100.00% (10/10)	100.00% (10/10)	83.33% (5/6)	80.00% (4/5)
contracts/periphery/registry/FulfillableRegistry.sol	100.00% (34/34)	97.37% (37/38)	89.47% (17/19)	92.86% (13/14)
All Files	**95,35%**	**95,10%**	**76,82%**	**87,05%**

We are grateful for the opportunity to work with the BANDO COOL S.A. DE C.V. team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the BANDO COOL S.A. DE C.V. team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo