Quantstamp Security Assessment Certificate

January 11th 2023 — Quantstamp Verified

Fabric CrowdFinancing

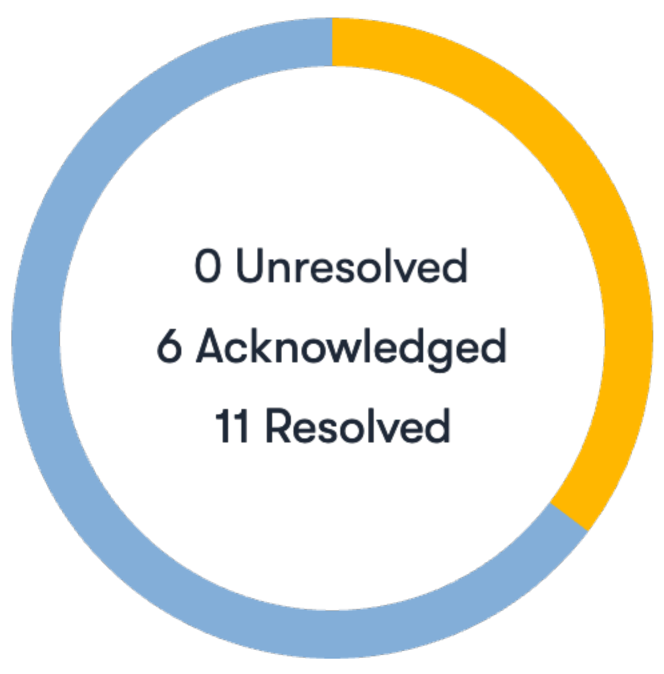
This audit report was prepared by Quantstamp, the leader in blockchain security.

Executive Summary

Type	CrowdFinancing
Auditors	Fatemeh Heidari, Security Auditor Ibrahim Abouzied, Auditing Engineer Ruben Koch, Auditing Engineer II Gelei Deng, Auditing Engineer II
Timeline	2022-12-05 through 2023-01-11
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Specification
Documentation Quality	Medium
Test Quality	Medium
Source Code	

Repository	Commit
Fabric CrowdFinancing	ecb40d6
withfabricxyz/contracts	e7ec5f9 fixes

Total Issues	**17** (11 Resolved)
High Risk Issues	0 (0 Resolved)
Medium Risk Issues	**2** (2 Resolved)
Low Risk Issues	**8** (7 Resolved)
Informational Risk Issues	**6** (2 Resolved)
Undetermined Risk Issues	**1** (0 Resolved)



High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational	The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
Undetermined	The impact of the issue is uncertain.

Unresolved	Acknowledged the existence of the risk, and decided to accept it without engaging in special efforts to control it.
Acknowledged	The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).
Fixed	Adjusted program implementation, requirements or constraints to eliminate the risk.
Mitigated	Implemented actions to minimize the impact or likelihood of the risk.

Summary of Findings

Fabric is a decentralized CrowdFinancing platform that enables groups of individuals to finance a project and receive collective rewards for their investments. The contract behaves differently depending on whether the state of the fundraiser is actively seeking funding, completed its fundraising, or has failed to meet its fundraising goals. If the minimum funding goal is reached, the money is sent to the beneficiary and, if applicable, a fee collector. If the minimum funding target is not reached by the end of the funding period, customers will be permitted to withdraw their initial donations. If the funding is successful, Eth or tokens are sent to the contract to pay back contributors in proportion to their initial contributions. It is worth mentioning that the contract cannot force the beneficiary to follow through with their obligations and pay back the shareholders

During the audit, we discovered several issues, most of which were of medium or low severity. We recommend that Fabric address them. In addition, tests for specific scenarios appear to be absent. We strongly recommend adding more test cases and increasing the branch coverage of the tests.

Update: Fabric team fixed most of the issues. The coverage is still low, and we recommend improving test coverage metrics and creating a public system specification.

ID	Description	Severity	Status
QSP-1	Use of Rebasing and Fee-Taking Erc20 Tokens Could Lead to Stuck Funds	^ Medium	Fixed
QSP-2	Contract Can Receive Payments Before the `FUNDED` State	^ Medium	Fixed
QSP-3	Improper Deposit Amount Restriction	⌄ Low	Fixed
QSP-4	Same Fee Collector and Depositor	⌄ Low	Fixed
QSP-5	Using `transfer()` for ETH May Fail in The Future	○ Informational	Acknowledged
QSP-6	The `Payoutsmadeto()` Function Could Potentially Revert Due to Overflows	⌄ Low	Mitigated
QSP-7	Missing Input Validation	⌄ Low	Fixed
QSP-8	Unlocked Pragma	⌄ Low	Fixed
QSP-9	Transfers Are Done with Approval Amounts Rather than Inputs	⌄ Low	Fixed
QSP-10	The Fee Collector Can Be Initialized to Withdraw Unlimited Fees	⌄ Low	Fixed
QSP-11	Direct Transfers to the Logic Contract During Funding Will Be Considered as Payout	⌄ Low	Acknowledged
QSP-12	Enable Processing of Funds as Soon as `_Fundtargetmax` Is Reached	○ Informational	Fixed
QSP-13	Payable Fallback Function in `Ethcrowdfinancingv1` Could Call Deposit During Funding State	○ Informational	Acknowledged
QSP-14	Upgradability	○ Informational	Acknowledged
QSP-15	Block Timestamp Manipulation	○ Informational	Acknowledged
QSP-16	Unnecessary Precision for Payout Calculation	○ Informational	Fixed
QSP-17	Unclear Payback Risk	? Undetermined	Acknowledged

Quantstamp Audit Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

DISCLAIMER:
If the final commit hash provided by the client contains features that are not within the scope of the audit or an associated fix review, those features are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

The Quantstamp auditing process follows a routine series of steps:

1. Code review that includes the following
   i. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.9.1

Steps taken to run the tools:

1. Install the Slither tool: `pip3 install slither-analyzer`
2. Run Slither from the project directory: `slither .`

Findings

QSP-1 Use of Rebasing and Fee-Taking Erc20 Tokens Could Lead to Stuck Funds

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `ERC20CrowdFinancingV1.sol`

Description: There are established ERC20 tokens that have mechanisms in place for balance modifications of accounts outside of regular token transfers. An example is Ampleforth, which rebases all balances every 24h based on market demands to sustain USD parity. Other ERC20 tokens, e.g. PAXG, deduct a transfer fee from the to-be-transferred amount.
Currently, the logic contract caches balance based on the set allowance from a user and not the transferred tokens; this could lead to problems with fee-taking tokens. Properly handling rebasing tokens is even trickier, as the contract's balance could change without any interactions from users.

Exploit Scenario: 1. Some depositors deposit some amount of the specified ERC20 token with either a deducting fee or a rebasing functionality. The `fundTargetMin` is eventually reached.

1. In the case of a fee-deducting token, a higher amount of tokens is cached in `_depositTotal` and in the `_deposits` mapping for each user than was received by the logic contract. In the case of a rebasing token, imagine that until `endTimestamp` is reached, the token balance in the logic's contract is reduced based on the rebasing

mechanism of the protocol.

2. A call to `processFunds()` after reaching `endTimestamp` attempts to transfer in total `_depositTotal` amount of tokens to the optional fee collector and the beneficiary. However, the transfers will fail due to the contract having a lower balance than `_depositTotal` and the funds become stuck. Only by directly transferring additional token to set the contract's balance to at least `_depositTotal`, the funds become unstuck.

The same problem could also arise in a failed funding phase, where the last depositors to withdraw will be unable to do so due to the contract having insufficient funds.

Recommendation: Instead of caching the allowance for deposits and payments, cache the difference in the contract balance before and after the transfer. There could be measures taken to support rebasing tokens, however, they would add significant complexity to the code. Provide documentation for campaign creators to warn them of using rebasing ERC20 tokens.

Update: The Fabric team fixed the issue in commit `c3fc5d8` by implementing the suggestion for fee-taking tokens and acknowledging that more external docs are required to warn against rebasing tokens.

QSP-2 Contract Can Receive Payments Before the `FUNDED` State

Severity: Medium Risk

Status: Fixed

File(s) affected: `ERC20CrowdFinancingV1.sol`

Description: The `makePayment()` function requires the contract to be in the `FUNDED` state before transferring payments from the user to the contract. However, a user can transfer tokens directly to the contract without the use of this function. Tokens received outside of the `makePayment()` function would still be paid out through `withdrawPayout()` since payouts are calculated based on the token balance of the contract.

Recommendation: Throughout the contract, track the amount of payments received through a `payments` variable that is incremented in `makePayment()` rather than the token balance of the contract.

Update: The Fabric team fixed the issue in commit `0b8a890` and also adjusted the documentation. The Fabric team acknowledged that the risk of accidental direct transfers results in locked funds.

QSP-3 Improper Deposit Amount Restriction

Severity: Low Risk

Status: Fixed

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Description: The minimum deposit amount `minDeposit` in both `EthCrowdFinancingV1.sol` and `ERC20CrowdFinancingV1.sol` should be restricted because the current design may lead to the situation where a funding may never be completed. If `_minDeposit > (_fundTargetMax-_fundTargetMin)`, the funding may never be completed due to max target restriction.

Exploit Scenario: Assume a simple scenario where the `_fundTargetMax` is 100 and `_fundTargetMin` is 80. If minimum deposit amount is set to 30 and the current funding balance is 75, then the final deposit cannot be accepted because such deposit will result in a total funding amount higher than targeted max. A malicious user can always deposit some amount of tokens/ETH to make the funding fail in this case.

Recommendation: Add restriction to ensure that `_minDeposit < (_fundTargetMax-_fundTargetMin)`.

Update: The Fabric team fixed the issue in commit `1f145c6`.

QSP-4 Same Fee Collector and Depositor

Severity: Low Risk

Status: Fixed

File(s) affected: `EthCrowdFinancingV1.sol`, `ERC20CrowdFinancingV1.sol`

Description: It is unclear whether the `_feeCollector` can be a depositor. If so, the function `allocateFeePayout()` will cause a logic error because it assumes that `_deposits` do not have the `_feeCollector` key.

Exploit Scenario: If the fee collector decides to deposit, `processFunds()` will overwrite the fee collector's previous deposits, making the previously allocated shares unclaimable.

Recommendation: Require `_feeCollector` not to participate in the deposition or modify the `allocateFeePayout()` function as `_deposits[_feeCollector] += feeAllocation;`.

Update: The Fabric team fixed the issue in commit `911d930`

QSP-5 Using `transfer()` for ETH May Fail in The Future

Severity: Informational

Status: Acknowledged

File(s) affected: `EthCrowdFinancingV1.sol`

Description: The functions `address.transfer()` and `address.send()` assume a fixed amount of gas to execute the call. The use of these functions protects against reentrancy attacks. The default amount of gas, however, may change in the future. In the worst case, it could lead to failed transfers.
However, more importantly, for smart contracts with a slightly more complex payable fallback function, every attempt to withdraw payouts could result in failed transfer due to an out-of-gas-exception, therefore the payouts become unredeemable.

Recommendation: We want you to be aware of this possibility. Whereas we do not recommend taking any action now, if you need more flexibility regarding the forwarded gas, you can use `call()` to perform transfers.

QSP-6 The `Payoutsmadeto()` Function Could Potentially Revert Due to Overflows

Severity: Low Risk

Status: Mitigated

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Description: The `payoutsMadeTo()` functions of both the `ERC20CrowdFinancingV1` and the `EthCrowdFinancingV1` use a high precision value of `1e18`. The precision value in both methods is used in `_deposits[account] * 1e18 * payoutTotal()`, which in some scenarios might lead to an overflow, since all the involved variables most likely contain high-digit values. The `uint256` type can reach values up to, `2^256 - 1`, which in scientific notation equals `1.1579209e+77`. If e.g. `deposits[account]= 1e20` and `payoutTotal = 1e40`, we could cause an overflow, since `_deposits[account] * 1e18 * payoutTotal() = 1e78 > type(uint256).max`. If such an overflow would occur, the user would be unable to withdraw their payouts.

Recommendation: Consider adding a check for an overflow before performing the calculation with such high precision.

Update: The Fabric team mitigated the issue in commit `fb0df39` by removing the redundant precision. But overflow is still possible.

QSP-7 Missing Input Validation

Severity: Low Risk

Status: Fixed

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Related Issue(s): SWC-123

Description: It is important to validate inputs, even if they only come from trusted addresses, to avoid human error.

In `ERC20CrowdFinancingV1` and `EthCrowdFinancingV1` contracts the `feeUpfrontBips` and `feePayoutBips` parameters passed to the `initialize` function is supposed to be between 0 and 10000.

This would result in `allocateFeePayout()` or `calculateUpfrontFee()` returning a value greater than `_depositTotal`, which would cause an underflow in the `processFunds()` function, causing the entire transaction to revert. So, as the funds can not be processed, the funds would become stuck indefinitely, as the contract's state would be unchangeable from `State.FUNDING`, enabling neither the withdrawal of deposits nor the transfer to the beneficiary.

The `initialize()` functions are missing input validation for the `startTimestamp` parameter to be greater than `block.timestamp()`.

In the `initializer()` function of both the `ERC20CrowdFinancingV1` and the `EthCrowdFinancingV1`, contracts are missing input validation that `minDeposit` is greater than zero.

Recommendation: We recommend adding the relevant checks.

Update: The Fabric team fixed the issue in commit `2deaae9`.

QSP-8 Unlocked Pragma

Severity: Low Risk

Status: Fixed

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Related Issue(s): SWC-103

Description: Every Solidity file specifies in the header a version number of the format `pragma solidity (^)0.8.*`. The caret (^) before the version number implies an unlocked pragma, meaning that the compiler will use the specified version *and above*, hence the term "unlocked".

Recommendation: For consistency and to prevent unexpected behavior in the future, we recommend to remove the caret to lock the file onto a specific Solidity version.

Update: The Fabric team fixed the issue in commit `9fcfe13`.

QSP-9 Transfers Are Done with Approval Amounts Rather than Inputs

Severity: Low Risk

Status: Fixed

File(s) affected: `ERC20CrowdFinancingV1.sol`

Description: `deposit()` and `makePayment()` transfer the full allowance of a user rather than an through a parameter of the desired transferred amount. Users may transfer an undesired amount of funds if the desired transferred amount may differ from the approval amount.

Recommendation: Update `deposit()` and `makePayment()` to take an `amount` parameter so that users always know how much they are transferring.

Update: The Fabric team fixed the issue in commit `cc9910e`.

QSP-10 The Fee Collector Can Be Initialized to Withdraw Unlimited Fees

Severity: Low Risk

Status: Fixed

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Description: The contracts optionally allow a `_feeCollector` to collect fees. However, `initialize()` permits a `_feeCollector` address value of zero while `_feeUpfrontBips` or `_feePayoutBips` are greater than zero, and vice versa.
Furthermore, there is also no constraint on the fee values. It is possible for the fees to consume 100% of the deposits from the beneficiary or to indefinitely dilute the current stakeholder shares.

Recommendation: Require a fee to be set if fees are enabled (`_feeCollector` is non-zero). Do not allow fees to be set if there is no `_feeCollector`. Enforce constraints on the maximum fee values.

Update: The Fabric team fixed the issue in commit `2deaae9`.

QSP-11 Direct Transfers to the Logic Contract During Funding Will Be Considered as Payout

Severity: Low Risk

Status: Acknowledged

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Description: Direct token transfers to the logic contract are actually listed in the documentation as an intended way for the beneficiary or other addresses to make a payment. However, direct transfers of tokens before successful funding will be immediately redeemable by the depositors after a state change to `State.FUNDED`, as from this point on the token balance of the contract is considered the amount of payments made. So, while direct transfers from the beneficiary are not an issue (though no `DEPOSIT` event will be emitted in that case), they are problematic for depositors. In case of failed funding, the tokens become stuck indefinitely, in case of successful funding, they will be distributed to properly registered depositors. The `EthCrowdFinancingV1` contract can somewhat mitigate this, as it only accepts payments if `_state == State.FUNDED`, however, it too can (mainly theoretically) receive additional ETH via a `selfdestruct` call of a contract with the logic contract address is the recipient of its funds.

Recommendation: Educate the end users with dedicated documentation on the potential pitfalls of direct transfers.

Update: The Fabric team acknowledged the issue:

> *We can't detect direct token transfers. Our user facing tools will not leverage this, and we will warn users this should not be done.*

QSP-12 Enable Processing of Funds as Soon as `_Fundtargetmax` Is Reached

Severity: Informational

Status: Fixed

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Description: Currently, the `processFunds()` functions always require the raise window to be over. However, as in that case, nothing remains to be done but wait, the UX could perhaps be improved by enabling the processing of funds during the raise window if the `_fundTargetMax` is already reached.

Recommendation: Consider if this is a net benefit; if so, adjust the `processFunds()` function accordingly. Some special care needs to be taken in that case, as e.g. the `expired()` function would also need to be modified.

Update: The Fabric team fixed the issue in commit `d815203`.

QSP-13 Payable Fallback Function in `Ethcrowdfinancingv1` Could Call Deposit During Funding State

Severity: Informational

Status: Acknowledged

File(s) affected: `EthCrowdFinancingV1.sol`

Description: Direct transfers of ETH currently get rejected, if they are not taking place after successful funding. The function could however simply call `deposit()` internally if the contract is currently in the funding stage, enabling a perhaps more simple interaction with the contract for end users. This is however not possible for arbitrary ERC20 tokens, so the resulting functional inconsistency between the `EthCrowdFinancingV1` and the `ERC20CrowdFinancingV1` contracts might not be desired.

Recommendation: Consider adjusting the fallback function to perform call to the `deposit()` function if the contract state is set to `State.FUNDING`.

Update: The Fabric team acknowledged the issue:

> *This would be a change for the worse. Imagine the scenario where a user sees they can deposit, so a transaction is prepared, and then sent... only to find out the contract was processed in the mean time and now the user has just made a payout to all depositors.*

We think that `EthCrowdFinancingV1` contract could also be adjusted to only distribute funds received via `makePayment()`; that way, direct-transfers would be rejected after the funding stage, not resulting in the loss of funds described in the above scenario.

QSP-14 Upgradability

Severity: Informational

Status: Acknowledged

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Description: While upgradability is not a vulnerability in itself, token holders should be aware that the token contract can be upgraded at any given time. This audit does not guarantee the behavior of future contracts that the token may be upgraded to.

Recommendation: The fact that the contract can be upgraded and reasons for future upgrades should be communicated to users beforehand.

Update: The Fabric team acknowledged the issue:

> *This will be addressed with documentation.*

QSP-15 Block Timestamp Manipulation

Severity: Informational

Status: Acknowledged

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Related Issue(s): SWC-116

Description: Projects may rely on block timestamps for various purposes. However, it's important to realize that validators individually set the timestamp of a block, and attackers may be able to manipulate timestamps for their own purposes. If a smart contract relies on a timestamp, it must take this into account.

Update: The Fabric team acknowledged the issue:

> *The use of time in this contract is important, but not critical.*

QSP-16 Unnecessary Precision for Payout Calculation

Severity: Informational

Status: Fixed

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Description: The `payoutsMadeTo` function in both `EthCrowdFinancingV1.sol` and `ERC20CrowdFinancingV1.sol` calculates the payouts and tries to increase precision by multiplying with `1e18` and then dividing by the same amount. However, such solution does not increase precision in the provided use cases. It is also not clear if 1e18 is selected because of ETH-wei conversion.

Recommendation: Remove the multiplication/division, or select different numbers in the numerator/denominator to provide better precision. To the best of our understanding, this additional precision is not needed.

Update: The Fabric team fixed the issue in commit `fb0df39`.

QSP-17 Unclear Payback Risk

Severity: *Undetermined*

Status: Acknowledged

File(s) affected: `ERC20CrowdFinancingV1.sol`, `EthCrowdFinancingV1.sol`

Description: The contracts are designed for groups of people to collectively finance a project with the option of receiving returns on their investment. However, it is stated in the documentation that the beneficiary is trusted but it is not documented the potential risks that the beneficiary may fail to pay back. Furthermore, there's no restriction on the beneficiary's behaviour.

Exploit Scenario: The beneficiary may never return the payback, and depositors suffer from investment loss.

Recommendation: Declare the potential risks in the notice/documentation clearly to users.

Update: The Fabric team acknowledged the issue:

> *This will be paired with closer-to-user UX that will convey the risks involved.*

Automated Analyses

Slither

Version 0.9.1 of Slither was used and found 19 results, most of which were either duplicates or false positives.

Adherence to Specification

1. In the documentation, it is declared that *If the fund target is met within the fund raising window, then processing the funds will transfer all raised funds to the beneficiary, minus optional fee, and change the state of the contract to allow for payouts to occur*. However, in the implementation, the function `processFunds()` can only be called when the raise window is expired. The fund cannot be processed even if the target is met. This applies to both ETH fund raising (`EthCrowdFinancingV1.sol#L244`) and ERC20 fundraising (`ERC20CrowdFinancingV1.solL#257`).
2. It seems that in both `EthCrowdFinancingV1.sol` and `ERC20CrowdFinancingV1.sol`, the maximum funding period is 90 days (7776000 seconds). However, this was not mentioned in both documentation and in-line annotation. It is recommended to provide an explanation of such design features.
3. In both `EthCrowdFinancingV1.sol#L150` and `ERC20CrowdFinancingV1.sol#160`, there are minor misalignments between code and annotation. Specifically, in `require(fundTargetMin > 0, "Min target must be >= 0");`, it should be modified as `"Min target must be > 0"`.
4. 2. In `ERC20CrowdFinancingV1.md` and the `README`, it stated that "Only the last step is required for deploying subsequent campaigns". However, further above in the documentation it is stated that "The CrowdFunding contract is deployed as a logic contract, and proxies are deployed for each campaign.". So, it is the last two steps that are required for deploying subsequent campaigns, as each campaign will have its own proxy deployed.
5. Typo "contraints" in `.sol#25` and `ERC20CrowdFinancingV1.md#21`.
6. Typo "transfered" in `ERC20CrowdFinancingV1.sol#254` and `EthCrowdFinancingV1.sol#241`.
7. Typo in `ERC20CrowdFinancingV1.md`, "Exammple".
8. Typo "expiired" in `ERC20CrowdFinancingV1.sol#475` and `EthCrowdFinancingV1.sol#459`.
9. In `ERC20CrowdFinancingV1.sol#194` and `EthCrowdFinancingV1.sol#183`, the documentation states that the value transferred "must be >= minimum fund amount". However, it should be `must be >= minimum deposit amount`. Also, in the same line in `ERC20CrowdFinancingV1.sol`, `msg.value` is being referred to, but it should be referring to the token allowance.

Adherence to Best Practices

1. More test cases should be created to cover more complex deposit logic for system testing. For instance, multiple depositors deposit multiple times.
2. Most functions in both `EthCrowdFinancingV1.sol` and `ERC20CrowdFinancingV1.sol` follow the same logic and use the same functions. Consider combining ETH native token as a special case into `ERC20CrowdFinancingV1.sol`.
3. Declare contract-level variables as `immutable` if they are only ever assigned in the constructor. All the stated variables should be declared as `immutable` in both the `ERC20CrowdFinancingV1` and the `EthCrowdFinancingV1` contract:
   - `_beneficiary`
   - `_fundTargetMin`
   - `_fundTargetMax`
   - `_minDeposit`
   - `_maxDeposit`
   - `_startTimestamp`
   - `_expirationTimestamp`

    - `_token` (only relevant for ERC20)

4. Functions marked as `public` that are not accessed from within the contract can be marked `external`. All the stated functions should be declared as `external` in both the `ERC20CrowdFinancingV1` and the `EthCrowdFinancingV1` contract (except for `tokenAddress()`):
    - `initialize()`
    - `deposit()`
    - `ownershipPPM()`
    - `depositAmount()`
    - `depositTotal()`
    - `processFunds()`
    - `returnOnInvestment()`
    - `withdraw()`
    - `minimumDeposit()`
    - `maximumDeposit()`
    - `startsAt()`
    - `expiresAt()`
    - `beneficiaryAddress()`
    - `tokenAddress()` (only relevant for ERC20)
    - `minimumFundTarget()`
    - `maximumFundTarget()`
5. Error message in `ERC20CrowdFinancingV1.sol#160` and `EthCrowdFinancingV1.sol#150` should be `>0` not `>= 0`.
6. In`ERC20CrowdFinancingV1`, the beneficiary is unnecessarily declared with the `payable` keyword. Consider removing it.
7. In `ERC20CrowdFinancingV1.sol#205` and `EthCrowdFinancingV1.sol#194`, `require(amount >= _minDeposit)` makes more sense than `require(total >= _minDeposit)`, since the first deposit already has to suffice the check. While both checks are functionally identical, consider changing it just for improved readability.
8. Misleading function name for `depositAmount()` in both the `ERC20CrowdFinancingV1` and the `ETHCrowdFinancingV1` contracts. The function is only returning the deposited amount of the user, yet the name could imply that it is a function meant to transfer a deposit of an implied amount of tokens.
9. The `allocateFeePayout()` functions should return zero if the fees are not set, not simply return and implicitly use the zero value in memory.
10. In `ERC20CrowdFinancingV1.sol#205` and `EthCrowdFinancingV1.sol#194`, `require(amount >= _minDeposit)` makes more sense than `require(total >= _minDeposit)`, since the first deposit already has to suffice the check. While both checks are functionally identical, consider changing it just for improved readability.
11. Missing test case for case of both activated front up fee and payout fee.
12. Extremely high `delta` parameter in the `testPayoutFees()` test cases of both contracts. Consider choosing a more reasonable value, as rounding errors that high would not be acceptable.
13. To improve readability and lower the risk of introducing errors when making code changes, it is advised to not use magic constants throughout code, but instead declare them once (as constant and commented) and use these constant variables instead. Following instances should therefore be changed accordingly:
    1. `ERC20CrowdFinancingV1.sol: #L159`: `7776000`, `L228`: `1_000_000`, `L293`, `L306` : `10_000`, `L364`: `1e18`.
    2. `EthCrowdFinancingV1.sol #L149`: `7776000`, `L215`: `1_000_000`, `L280`,`L293` : `10_000`, `L348`: `1e18`.

Test Results

Test Suite Results

Of the provided 56 tests, all were successfully executed. We recommend that more test cases be created to cover complex deposit logic for system testing. For instance, multiple depositors deposit multiple times. Test case for case of both activated front up fee and payout fee are missing.

```
Running 26 tests for test/finance/EthCrowdFinancingV1.t.sol:EthCrowdFinancingV1Test
[PASS] testDeposit() (gas: 80949)
[PASS] testDepositAfterFailed() (gas: 166297)
[PASS] testDepositAfterFunded() (gas: 203669)
[PASS] testDepositWithNoBalance() (gas: 18114)
[PASS] testDoubleWithdraw() (gas: 264467)
[PASS] testEarlyProcess() (gas: 274090)
[PASS] testEarlyWithdraw() (gas: 13416)
[PASS] testEmptyDeposit() (gas: 26659)
[PASS] testEndChecks() (gas: 24228)
[PASS] testFundingFailure() (gas: 166480)
[PASS] testFundsTransfer() (gas: 201515)
[PASS] testInitialDeployment() (gas: 32182)
[PASS] testInvalidFeeConfig() (gas: 1233889)
[PASS] testLargeDeposit() (gas: 35633)
[PASS] testManyDeposits() (gas: 101414)
[PASS] testManyDepositsFromMany() (gas: 152640)
[PASS] testMultiReturns() (gas: 286564)
[PASS] testPayoutFees() (gas: 1775858)
[PASS] testReturns() (gas: 229188)
[PASS] testSecondPassFail() (gas: 158247)
[PASS] testSecondProcess() (gas: 195595)
[PASS] testSmallDeposit() (gas: 33601)
[PASS] testStartChecks() (gas: 19890)
[PASS] testUpfrontFees() (gas: 1598257)
[PASS] testUpfrontFeesSplit() (gas: 1598098)
[PASS] testWithdraw() (gas: 272529)
Test result: ok. 26 passed; 0 failed; finished in 6.01ms

Running 30 tests for test/finance/ERC20CrowdFinancingV1.t.sol:ERC20CrowdFinancingV1Test
[PASS] testAllowanceMismatch() (gas: 66593)
[PASS] testDeposit() (gas: 127888)
[PASS] testDepositAfterFailed() (gas: 236956)
[PASS] testDepositAfterFunded() (gas: 259807)
[PASS] testDepositWithNoBalance() (gas: 31226)
[PASS] testDoubleWithdraw() (gas: 368114)
[PASS] testEarlyProcess() (gas: 377709)
[PASS] testEarlyWithdraw() (gas: 13283)
[PASS] testEndChecks() (gas: 24198)
[PASS] testFundingFailure() (gas: 248700)
[PASS] testFundsTransfer() (gas: 263497)
[PASS] testInitialDeployment() (gas: 37390)
[PASS] testInvalidFeeConfig() (gas: 1507439)
[PASS] testLargeDeposit() (gas: 63108)
```

```
[PASS] testManyDeposits() (gas: 171152)
[PASS] testManyDepositsFromMany() (gas: 235192)
[PASS] testMultiReturns() (gas: 408874)
[PASS] testPayoutFees() (gas: 2188547)
[PASS] testProfit() (gas: 336142)
[PASS] testReinit() (gas: 18250)
[PASS] testReturns() (gas: 342111)
[PASS] testReturnsViaPayoutFn() (gas: 341415)
[PASS] testSecondPassFail() (gas: 233912)
[PASS] testSecondProcess() (gas: 256744)
[PASS] testSmallDeposit() (gas: 60993)
[PASS] testStartChecks() (gas: 19903)
[PASS] testUnapprovedDeposit() (gas: 31228)
[PASS] testUpfrontFees() (gas: 1968930)
[PASS] testUpfrontFeesSplit() (gas: 1967800)
[PASS] testWithdraw() (gas: 377443)
Test result: ok. 30 passed; 0 failed;
```

Code Coverage

File	% Lines	% Statements	% Branches	% Funcs
src/finance/ERC20CrowdFinancingV1.sol	100.00% (118/118)	100.00% (129/129)	73.08% (57/78)	100.00% (34/34)
src/finance/EthCrowdFinancingV1.sol	**95.19% (99/104)**	**94.55% (104/110)**	**72.58% (45/62)**	**93.33% (28/30)**
Total	97.75% (217/222)	97.49% (233/239)	72.86% (102/140)	96.88% (62/64)

While the statement coverage reaches 97,17%, the branch coverage is only 75.45%. We highly recommend adding further tests to increase both coverage and in particular the branch coverage to at least 90% or ideally higher.

Update: The statement coverage is 97.49% and the branch coverage is still low 72.86% which is still low.

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

`8199e13e37af5843f5d3c6d1fbe5f38699d5e2d0432d9a7c50c6b7fb7c72ec3d ./finance/ERC20CrowdFinancingV1.sol`

`5e236d13875de160a7d4db5d1b65007d41ebfed21c6b360eea66203ee1a988b8 ./finance/EthCrowdFinancingV1.sol`

Tests

`05df4c180c1e5f75e3ee71faca5ea55577adf7cbbcb7130b6602960b52d3131b ./finance/EthCrowdFinancingV1.t.sol`

`a3dd6365282428d845dcff0b4d5a225c32170fc62d6ce58b284b5eac25fbc0d2 ./finance/ERC20CrowdFinancingV1.t.sol`

Changelog

- 2022-12-09 - Initial report
- 2023-01-11 - fix-review

About Quantstamp

Quantstamp is a global leader in blockchain security backed by Pantera, Softbank, and Commonwealth among other preeminent investors. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its white glove security and risk assessment services.

The team consists of web3 thought leaders hailing from top organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Many of the auditors hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 250 audits and secured over $200 billion in digital asset risk from hackers. In addition to providing an array of security services, Quantstamp facilitates the adoption of blockchain technology through strategic investments within the ecosystem and acting as a trusted advisor to help projects scale.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Aave, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT