verichains

SECURITY AUDIT OF

FUND ESCROW SMART CONTRACTS

Public Report

Sep 11, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Sep 11, 2024. We would like to thank the Staria for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Fund Escrow smart contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Fund Escrow smart contracts

Staria Swisspad unlocks exclusive access to early-stage venture capital deals before they list on exchanges. Staria Launchpad guides investors through the crypto venture landscape with a meticulous deal flow, ensuring access to high-quality projects through a transparent, multi-phase selection process.

1.2. Audit Scope

This audit focused on identifying security flaws in the code and the design of the Fund Escrow smart contracts.

It was conducted on commit `9a9f99bd902064b03a7616806797fcf4e41d4315` from the git repository `https://github.com/StariaNetwork/staria-sc`.

The following files were made available in the course of the review:

SHA256 Sum	File
8503c951c84d35ffc92fd960da85afaedb95b051ccbd1de4a3c43e8b4cf1387c	./libraries/ErrorLibraryVRF.sol
9c09ad30e0cc38b8623fe6a11b1fba19011b76fefd8182e28f30fb41c92c0b4d	./libraries/ErrorLibrary.sol
72bd3db99ebb23d159c79539b5f06e9d8676bcf49880e17e01530f163963a386	./libraries/VrfPayload.sol
c3b00fe1a26043b3829de946187cc3be05397c8aaaffb1867dd9d782ef1bbc0d	./libraries/TransactionPayload.sol
dfee348b17af94f400789152c45710918cbdec83dde320b35876575e4d17a80a	./FundsEscrow/Escrow.sol
85c2b413395344926b105f37ff79e6dbe656e3c988cf442edbdcbfe82d399aa9	./FundsEscrow/RefundEscrow.sol
fe16c48b3a76bbb17c8388bd8789ae1d758845855a48d74e3d0fd741f3fdebf9	./ReferralEscrow/ReferralRefundEscrow.sol
37e1f054f7f650517aa1a476b4e07328f86b113766ccee0b8d2a2204878c526e	./ReferralEscrow/ReferralEscrow.sol
2ec4d9b2c7f94cc53ccbdae67384207dfb82a5f639f22ebb7ea7a082f19aa310	./Funding.sol

c3f09c5a1ea69bc4a2d9d1fafa92200e99d0d95f6d761e017466c6a0319a53a0	./VRFConsumerBase.sol
d71dc7170f2cbea96477f70a9992d7ec710f07f011db4fa89c0f7b063cc91aee	./VRFConsumerContract.sol

1.3. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.

SEVERITY LEVEL	DESCRIPTION
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Staria acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Staria understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Staria agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Staria will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Staria, the final report will be considered fully accepted by the Staria without the signature.

2. AUDIT RESULT

2.1. Overview

The Fund Escrow smart contracts was written in `Solidity` language. The source code was written based on OpenZeppelin's library.

2.1.1. Funding contract

This contract is designed to manage funding rounds, facilitating a structured process for raising and handling funds through multiple phases. Here's a breakdown of its core functionality:

- **Initiating Funding Rounds:**
  - A customer account, starts a new funding round. Each funding round is linked to a unique `RefundEscrow` contract to handle the funds.
- **User Participation and Deposits:**
  - The organizer, acting as a back-end wallet, creates a unique, signed payload for every registered user. This payload authorizes the user to deposit a specific amount of tokens into the funding round.
- **Funding Round Outcomes:**
  - The funding round concludes with one of two possible outcomes:
    - **Success:** The customer account finalizes the funding round. Following this, the beneficiary or any user can invoke `beneficiaryWithdrawToken()` from the `RefundEscrow` contract to withdraw the funds.
    - **Failure:** The customer account triggers a refund process, returning all tokens to the original contributors.

A single `ReferralRefundEscrow` contract is shared across all rounds to manage referral fees. This contract allows beneficiaries to withdraw accumulated referral fees from successful and closed funding rounds.

2.1.2. VRFConsumerContract contract

This contract designed to securely request and receive random numbers from a Verifiable Random Function (VRF) Coordinator. It ensures the randomness is verifiable and tamper-proof.

Users request random numbers by submitting a valid payload and signature. The signature is then validated to ensure the caller is authorized. The contract sends the request to the VRF Coordinator and stores the request details. When the random numbers are received from the VRF, they are stored, and corresponding events are emitted.

2.2. Findings

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

Staria fixed the code according to Verichains's draft report in commit `ad280b68c3ed09f84c663e0e2cb68213b7971046`.

#	Issue	Severity	Status
1	requestRandomWords can be spammed to drain all the subscription fund	HIGH	Fixed
2	The signature can be reused for all referrers to claim their referral fees	MEDIUM	Fixed
3	The softCap must be less than or equal the maxFundingAmount	MEDIUM	Fixed
4	High gas cost as the number of fundings grows	LOW	Fixed
5	resetAllBeneficiaries is called multiple times unnecessary	LOW	Fixed
6	Use `initialize` functions instead of `unchain`	INFORMATIVE	Fixed
7	Wrong dev docs in many places	INFORMATIVE	Fixed
8	Unusual logic for finalizing funding	INFORMATIVE	Acknowledged

2.2.1. requestRandomWords can be spammed to drain all the subscription fund HIGH

Affected files:

- VRFConsumerContract.sol

`requestRandomWords` function can be called by anyone so attacker can call this multiple times to drain all the subscription fund (the money which is used for requesting random words).

```
function requestRandomWords() external nonReentrant {
    // Will revert if subscription is not set and funded.
    uint256 requestIdReceive = coordinator.requestRandomWords(
        keyHash,
        subId,
        requestConfirmations,
        callbackGasLimit,
```

```
        numWords
    );

    requestsSentCount++;
    requestsSentRequestId[requestsSentCount] = requestIdReceive;
    requestsSentRequesterAddress[requestsSentCount] = msg.sender;
    emit RequestRandomWords(
        msg.sender,
        requestsSentCount,
        requestIdReceive
    );
}
```

RECOMMENDATION

Add authorization mechanism to restrict who can request random words.

UPDATES

- **Sep 6, 2024**: This issue has been acknowledged and fixed.

2.2.2. The signature can be reused for all referrers to claim their referral fees MEDIUM

Affected files

- Funding.sol

This function allows referrers to claim their referral fee using a signature from the organization. However, the issue is that the signature doesn't specify the referrer or the particular funding being claimed. As a result, once the signature is generated, all referrers can use that same signature to claim their referral fees for all current and future fundings.

If this is your intended behavior, we suggest making it simpler by switch a boolean flag in the contract to enable the claiming of referral fees.

The dev document should be updated to reflect the correct behavior of the function. This function is used to allow the referrer to claim their referral fees with organization allowance.

```
/**
 * @notice Allows the organizer to withdraw tokens on behalf of the beneficiary.
 * @param withdrawAllowed The withdrawal data structure.
 * @param signature The signature that authorizes the withdrawal.
 */
function beneficiaryWithdrawToken(
    TransactionPayload.Withdraw memory withdrawAllowed,
    bytes memory signature
) public {
    // Compute the hash of the withdrawal data using EIP-712 standard
    bytes32 digest = _hashTypedDataV4(
        keccak256(abi.encode(WITHDRAW_TYPEHASH, withdrawAllowed))
```

```
    );

    // Recover the signer's address from the signature
    address recoveredSigner = ECDSAUpgradeable.recover(digest, signature);

    // Ensure the recovered signer matches the organizer's address
    if (recoveredSigner != organizer) {
        revert ErrorLibrary.InvalidSignature();
    }
    if (withdrawAllowed.isAllowed) {
        // Allow the beneficiary to withdraw tokens
        referralRefundEscrow.beneficiaryWithdrawToken(msg.sender);
    }
}
```

UPDATES

- **Sep 6, 2024**: This issue has been acknowledged and fixed.

2.2.3. The softCap must be less than or equal the maxFundingAmount MEDIUM

Affected files:

- RefundEscrow.sol
- ReferralRefundEscrow.sol

`setDeadlineAndMaxFundingAmount` functions don't check that the `softCap` is less than or equal the `maxFundingAmount` like in the init functions. This could cause a wrong funding logic when the customer set wrong `maxFundingAmount`.

```
constructor(
    address beneficiary_,
    uint256 _softCap,
    uint256 _maxFundingAmount,
    uint256 _deadline,
    IERC20 erc20Token_
) {
    ...
    if(_softCap > _maxFundingAmount) {
        revert ErrorLibrary.incorrectSoftCap(_softCap,_maxFundingAmount);
    }
    ...
}

/**
 * @notice Sets the deadline and maximum funding amount for this escrow
     * @dev This function can only be called by the owner
     * @param _maxFundingAmount The new maximum funding amount for the escrow.
     * @param _deadline The new deadline for the escrow.
 */
```

```
function setDeadlineAndMaxFundingAmount(
    uint256 _maxFundingAmount,
    uint256 _deadline
) external onlyOwner {
    maxFundingAmount = _maxFundingAmount;
    deadline = _deadline;
}

/**
 * @notice Sets the deadline and maximum funding amount for a specific escrow.
     * @dev This function can only be called by the owner and only for an existing escrow.
     * @param _escrowId The ID of the escrow to update.
     * @param _maxFundingAmount The new maximum funding amount for the escrow.
     * @param _deadline The new deadline for the escrow.
 */
function setDeadlineAndMaxFundingAmount(
    uint256 _escrowId,
    uint256 _maxFundingAmount,
    uint256 _deadline
) external onlyOwner onlyEscrowExists(_escrowId) {
    _escrowMaxTotalAmount[_escrowId] = _maxFundingAmount;
    _escrowDeadline[_escrowId] = _deadline;
}
```

RECOMMENDATION

Add check to ensure that the `softCap` is less than or equal the `maxFundingAmount`.

UPDATES

- **Sep 11, 2024**: This issue has been fixed.

2.2.4. High gas cost as the number of fundings grows LOW

Affected files:

- ReferralRefundEscrow.sol

This function loops over all fundings (including the unnecessary ones), which increases gas costs as the number of fundings grows. We can reduce wasted gas by allowing the referrer to specify an array of funding IDs that are available for claim.

```
function beneficiaryWithdrawToken(address _beneficiary) external onlyOwner {

    uint256 maxTokenPayment;
    for (uint256 i = 1; i <= escrowCounter; i++) {
        if (state(i) == State.Closed) {
            maxTokenPayment += _beneficiariesToken[i][_beneficiary];
            _beneficiariesToken[i][_beneficiary] = 0;
        }
```

```
    }

    if (maxTokenPayment > 0) {
        escrowToken.safeTransfer(_beneficiary, maxTokenPayment);
        emit BeneficiaryWithdrawToken(_beneficiary, maxTokenPayment);
    }
}
```

UPDATES

- **Sep 6, 2024**: This issue has been acknowledged and fixed.

2.2.5. resetAllBeneficiaries is called multiple times unnecessary LOW

Affected files:

- ReferralRefundEscrow.sol

The `refundTokenToAllPayees` function is called multiple times to refund payees in batches. However, in each batch, `resetAllBeneficiaries` is unnecessarily called again, leading to significant gas waste.

```
function refundTokenToAllPayees(
    uint256 _escrowId,
    uint256 _payeeRangeA,
    uint256 _payeeRangeB
) public virtual override onlyOwner {
    if (state(_escrowId) != State.Refunding) {
        revert ErrorLibrary.NotInRefundingState();
    }

    super.refundTokenToAllPayees(_escrowId, _payeeRangeA, _payeeRangeB);
    resetAllBeneficiaries(_escrowId);
}
```

RECOMMENDATION

Add a boolean flag to check if the beneficiaries have already been reset or move the reset logic to a separate function.

UPDATES

- **Sep 6, 2024**: This issue has been acknowledged and fixed.

2.2.6. Use init functions instead of unchain INFORMATIVE

Affected files:

- Funding.sol

The contract should use the `initialize` functions instead of `unchain` functions to initialize the contract. The function `__{ContractName}_init_unchained` found in every contract is the initializer function minus the calls to parent initializers, and can be used to avoid the double initialization problem, but doing this manually is not recommended. Only use them when the contract need to avoid the double initialization problem.

Refer here for more information:*https://docs.openzeppelin.com/contracts/4.x/upgradeable#multiple-inheritance*

For example, we just need to init `__Ownable2Step_init` here instead of manually init both `__Ownable_init_unchained` and `__Ownable2Step_init_unchained`.

```
function initialize(
    address _organizer,
    address _customer,
    IERC20Upgradeable _protocolEscrowFundingToken
) external initializer {
    __EIP712_init_unchained("Funding", "1.0");
    __Ownable_init_unchained();
    __Ownable2Step_init_unchained();
    __Pausable_init_unchained();
    organizer = _organizer;
    customer = _customer;
    protocolEscrowFundingToken = _protocolEscrowFundingToken;
    referralRefundEscrow = new ReferralRefundEscrow();
    _pause();
}
```

UPDATES

- **Sep 6, 2024**: The contract should use all the `initialize` functions except when the contract need to avoid the double initialization problem (which not in this case).

2.2.7. Wrong dev docs in many places INFORMATIVE

The dev docs is wrong in many places, for example:

"Function for the beneficiary to withdraw tokens when the contract is closed" => `beneficiaryWithdrawToken` function is used for the beneficiary to withdraw tokens when the funding is closed or the soft cap reached.

"Deposits tokens to a refundee if the contract state is Active" => should be "Deposits tokens from a refundee"

"Deploys a new instance of `ReferralRefundEscrow` for handling refund operations (`referralRefundEscrow`)." => should be "handling referral operations"

"Only the organizer can call this function to create a new funding escrow." => `createNewFunding` is only for customer, not organizer

Please double-check the dev docs of all functions to make sure it's correct, otherwise, some business logic could be misunderstood and we can't keep a right check for the audit.

UPDATES

- **Sep 6, 2024**: This issue has been acknowledged and fixed.

2.2.8. Unusual logic for finalizing funding INFORMATIVE

Affected files:

- Funding.sol

In current contract, funding can be set to `close` or `refund` state manually by the customer without any restriction. But in normal funding logic, the funding is only success if the soft cap reached, otherwise, the money is refunded to the investor. The current code does not restrict anything expect the deadline time so the customer can set the funding to `close` state (success funding) without reaching the soft cap.

Another case is when the soft cap reached, the customer can withdraw money to the beneficiary address, but he can also `enableRefunds` (failed funding) which prevent referrers from getting their referral fee. The referral tokens will then be stuck in the `ReferralRefundEscrow` contract, the investors will be unable to claim refund because there are no funds left in the `RefundEscrow` contract.

The dev document for `close` contract is wrong in "can close the specified funding by finalizing `refunds`". This function is used for finalizing the funding, not refunding.

```
/**
 * @notice Only the Multi-sig wallet 'customer' can close the specified funding by
finalizing refunds if the deadline has passed.
 *      Events are emitted in the referral escrow and the specific escrow contracts
 * @dev Checks if the current block timestamp is after the escrow deadline before
proceeding.
 *      Calls the close function in both the referralRefundEscrow and the specific escrow.
 * @param _fundingId The ID of the funding (escrow) to be closed.
 * - Retrieve escrow deadline from referralRefundEscrow
 * - Revert if the current block timestamp is before the escrow deadline
 * - Close the specific escrow by calling the close function
 * - Close the referral refund escrow by calling the close function
 */
function close(uint256 _fundingId) external onlyCustomer {
    (, , , , uint256 _escrowDeadline) = referralRefundEscrow.getEscrowInfo(
        _fundingId
    );
```

```
    if (block.timestamp < _escrowDeadline) {
        revert ErrorLibrary.CanOnlyCloseAfterDeadline(
            block.timestamp,
            _escrowDeadline
        );
    }
    escrow[_fundingId].close();
    referralRefundEscrow.close(_fundingId);
}

/**
 * @notice Only the Multi-sig wallet 'customer' can enable the refunding for the specified
funding after the escrow deadline has passed.
     *      Events are emitted in the referral escrow and the specific escrow contracts
     * @dev Checks if the current block timestamp is after the escrow deadline before
proceeding.
     *      Calls the enableRefunds function in both the referralRefundEscrow and the
specific escrow.
     * @param _fundingId The ID of the funding (escrow) to enable refunds for.
     */
function enableRefunds(uint256 _fundingId) external onlyCustomer {
    (, , , , uint256 _escrowDeadline) = referralRefundEscrow.getEscrowInfo(
        _fundingId
    );
    if (block.timestamp < _escrowDeadline) {
        revert ErrorLibrary.CanOnlyEnableRefundAfterDeadline(
            block.timestamp,
            _escrowDeadline
        );
    }
    escrow[_fundingId].enableRefunds();
    referralRefundEscrow.enableRefunds(_fundingId);
}
```

RECOMMENDATION

The contract need to restrict the finalization of the funding:

- The funding can only be set to `close` state if the soft cap reached.
- The funding can only be set to `refund` state if the soft cap is not reached.

UPDATES

- **Sep 6, 2024**: This issue has been acknowledged. The Staria stated that this is the intended behavior, allowing the `Customer` to manually set the funding status to `close` or `refund`.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Sep 6, 2024*	Private Report	Verichains Lab
1.1	*Sep 9, 2024*	Private Report	Verichains Lab
1.2	*Sep 11, 2024*	Private Report	Verichains Lab

Table 2. Report versions history