OpenZeppelin

OpenBazaar’s Escrow Audit

OPENZEPPELIN SECURITY | OCTOBER 25, 2018

Security Audits

The OpenBazaar team asked us in October 2018 to review and audit their Escrow contract, one of several in their framework. We audited the code, and shared the report privately with the OpenBazaar team. We are now, upon their request, making the audit report public.

The audited file is `Escrow_v1_0.sol`, at commit `c4f02cdd41cb85d28bba637a01f20a8ee8bb04b`. Additional information including the contract’s specification can be found here.

Here is our assessment and recommendations, in order of importance.

Critical Severity

None.

Transactions in the `Escrow_v1_0` contract are described by the `Transaction struct`. Among its fields, `timeoutHours` is used to set a timeout after which a seller can execute the transaction without the need for the buyer's or the moderator's signatures. In order to do this, the seller calls the `execute` function, which itself calls `verifyTransaction`, which finally calls `isTimeLockExpired`. This last function verifies that the time elapsed since the last modification of the transaction, tracked by the `lastModified` field in the `Transaction struct`, is greater than the transaction's time lock.

A malicious buyer trying to prevent a transaction from being executed can periodically update it by adding tiny amounts of Ether or tokens to its value through the `addFundsToTransaction` or `addTokensToTransaction` functions respectively. These calls will reset the `lastModified` field of the transaction, thus preventing it from ever reaching the time lock. If the transaction requires only two signatures, a seller can appeal to the moderator to execute it. If the transaction requires three signatures, however, the funds will be effectively locked until the buyer ceases to update it.

Consider keeping track of, and limiting, the number of times a transaction can be modified in order to prevent the indefinite locking of funds by buyers.

Medium Severity

Signature ordering may prevent transaction verification

In the `verifyTransaction` function, an `if clause` defines the conditions under which transactions fail to be verified (and the associated Ethereum transaction reverts). One way to have transactions verified is to call the execute function with a number of signatures above the transaction threshold. Another, meant as a *failsafe*, consists of having the transaction executed with just the seller's signature once the time lock is expired.

The implementation of the *failsafe* case requires the `execute` function to be called with the seller's signature as the last one in the signature component arrays for the transaction to be validated. This mismatch between specification and implementation entails cases where according to the specification the funds should be released, but they will not according to the implementation. One such case occurs when three signatures are required, the time lock is already expired, and

Furthermore, the error message provided will be "sig.length is under the threshold", which is not diagnostic of the actual failure (see **Wrong error messages on transaction verification failure** below).

Consider an implementation that is agnostic to signature ordering in order to fully match the specification.

Convoluted transaction verification implementation

`execute` is the function responsible for releasing funds to destination addresses, once all the necessary verification steps are taken. Currently, the entire process for verifying a transaction is divided and scattered throughout separate parts of the code, making it difficult for users to understand what requirements a transaction should meet in order for it to be approved, and rendering the code more error-prone and difficult to test.

From `execute`, which has a first `require` clause of its own, the `verifyTransaction` function is called. This function in turn call `verifySignatures`, which has several require statements, and then `isTimeLockExpired`, to finally reach an `if clause` that obscurely checks for further conditions using nested `and` and `or` operators and reverting when some of them are met. Back in `execute`, the `transferFunds` function is called within a require statement, which performs additional verifications distributed in several `require` clauses within loops.

Consider decoupling the verification of transaction requirements from the business logic as much as possible, encapsulating validations in specific straightforward functions that can be easily reused and tested.

Low Severity

Dead code in transferFunds function

Inside the `transferFunds` function, a multiple conditional is used to determine the transaction type (*i.e.* whether it is in Ether or in tokens). A `else` clause is included as a fallback to revert the Ethereum transaction in case the received transaction type does not match a known value. However, given that transaction types can only be set by the smart contract (see `L142` and `L194`), this `else` clause will never be executed.

Lack of explicit return statements

Several functions in the codebase have implicit return statements (`checkBeneficiary`, `calculateRedeemScriptHash`, `transferFunds`, `verifySignatures` and `isTimeLockExpired`).

Not using explicit return statements on functions that are expected to return values can lead to problems in future changes to the codebase. Should a return parameter's name be eventually removed (for instance if `returns (bool check)` is replaced with `returns (bool)`), no compilation-time warnings will be thrown by the Solidity compiler. In this scenario, the function's return value will be always automatically set to the default value of the return type, in this case `false`, which could lead to unforeseen issues.

Consider modifying the listed functions to include explicit return statements.

Unnecessary field in Transaction struct

The `scriptHash` field is being unnecessarily stored within the `Transaction struct`. Considering that it is never used other than as an index to access `Transactions` in mappings, this field can safely be removed from the `struct` to avoid duplication.

Wrong error message on transaction verification failure

The `verifyTransaction` function implements several checks to verify the conditions under which transactions should be accepted or rejected. One complex validation is performed in a single `if` `clause`, which has an associated error message that does not cover all possible causes of failure.

Consider refactoring the `if` clause in order to include specific error messages for each checked condition.

Misleading comment in `execute` function

The docstrings for `execute` function hints at moderators being the ones responsible for executing transactions. Yet, the function does not validate who the calling account actually is, meaning that execute can be called by anyone—including actors not involved in the transaction.

The error message in the `inFundedState` modifier mentions “dispute” as a possible transaction state. The `enum` in charge of defining transaction states, however, only lists “FUNDED” and “RELEASED” as possible states.

Consider either including this third state in the `enum`—along with the necessary logic to support it—or suitably modifying the error message.

Inconsistent coding style

There is a significant coding style difference between code segments:

- The wrapping of statement structures varies widely (as seen in events `L29` and `L35`).
- Indentation in `if` `clauses` is inconsistent.
- There is inconsistent spacing between typing, such as in lines: `49-51`, `54` and `59`.
- There is no ruling standard defining line wrapping (e.g. line `241`). Recommended is 80 columns.
- There is an inconsistent ordering of visibility and custom modifiers (e.g. `addTransaction` and `addFundsToTransaction`).

Consider following best practices and applying the same style guidelines across the codebase (see Solidity’s style guide for reference).

Grammatical errors in code and comments

Many comments are poorly written, with some containing grammatical or typing errors (e.g. “*desgined*”, “*transaction does not exists*”, “*singatures*”). This extends to the “*transactionDoesNotExists*” modifier name, which also has a grammatical error.

Consider fixing these to improve code readability.

Notes & Additional Information

- The `transferFunds` function is in charge of transferring funds to the corresponding beneficiaries when executing a transaction. In the case of Ether transactions, this is achieved via the *push-payment* pattern (i.e. a `transfer` is done to each of the destination pattern is accepted. Nonetheless, it is still relevant to emphasize that a *pull-payment* strategy should be further studied and considered, since the current implementation might lead to the lockout of funds sent to the Escrow contract. In the case such strategy is pursued, contemplate building upon OpenZeppelin's `PullPayment` contract.

- Consider indexing the `scriptHash` argument in all events to allow users to search and filter events using this field.
- The function `isTimeLockedExpired` returns `false` when the parameter `timeoutHours` from the transaction equals 0. This implies an *infinite* timeout, as opposed to an *immediate* one. Ensure that this is expected behavior and properly document it.
- verifySignatures is provided with the `r`, `s`, and `v` outputs of the ECDSA signatures trying to execute a transaction, and then verifies each one of them. Consider using the thoroughly tested OpenZeppelin's ECRecovery library to avoid the need of splitting the signatures before the call and improving code legibility.
- In both `addTokensToTransaction` and `addFundsToTransaction` a local variable `_value` is defined to track the transaction value. In neither case is this definition necessary, with the value already tracked by the parameter that is passed to the function in the first case (`value`), and by the `msg.value` variable in the second.
- `addTokensToTransaction` uses both `msg.sender` and `transactions[scriptHash].buyer` to represent the buyer interchangeably. Consider using only `msg.sender` (which is already used in `addFundsToTransaction`), and adding a comment on each function's docstring about this.
- In `_addTransaction`, after a `Transaction` has been created and stored, the *owners* are also added by setting the `isOwner` attribute of the current transaction. On line `634`, the `moderator` address is set as owner even if it is equal to `address(0)` (no moderator case). Consider checking for the moderator's address validity before registering it as owner.
- Consider prefixing all `private` and `internal` functions with an underscore to clearly denote their visibility (e.g. `verifyTransaction` and `transferFunds`).
- If `public` state variables are not expected to be read from outside the contract, consider restricting their visibility to private, in particular if getters to read such data are already defined (e.g. `getAllTransactionsForParty`).

required to release funds) would benefit from a name that better reflects what the variable represents.

- There are code segments where a single `require` clause includes multiple conditions. Consider isolating each condition in its own `require` clause, and having more specific error messages for each.
- Parameter order in `addTransaction` and `addTokenTransaction` is different from each function's documented `@param` list. Consider documenting parameters in the same order as they are defined in each function's signature
- `calculateRedeemScriptHash`'s docstring states how the script hash is to be obtained. While the Escrow contract's address is being used in the hash calculation, it is currently missing from the docstring, consider adding it.
- The contract's docstring would benefit from a more thorough and clearer description of its purpose and use cases.
- Consider adding a comment specifying the time units in which the `lastModified` field of a transaction is measured.
- Even though test files were out of the audit's scope, they were used as a guide to develop and test specific behaviors during the audit. In them, tests with wrong messages in asserts were found (*"Transaction was sent from buyer's address"*, where it should be "*was not sent*"). Furthermore, two test cases, `L149` and `L198`, with the same name but different content were found. Consider ensuring all messages correspond with what is being evaluated, and expanding test case descriptions to be more precise.

Conclusion

No critical and one high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.

Related Posts

Beefy Zap Audit

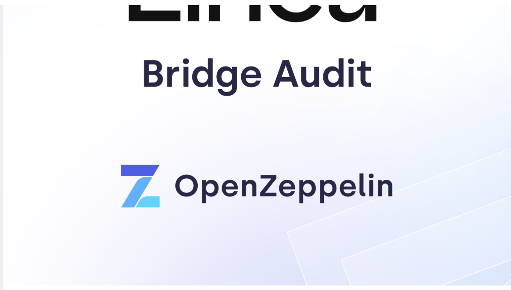
BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits



Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits