OpenZeppelin

Freeverse Payments Contract Audit

OPENZEPPELIN SECURITY | JUNE 16, 2022

Security Audits

This security assessment was prepared by **OpenZeppelin**, protecting the open economy.

Table of Contents

Summary

Type
    DeFi
Timeline
    From 2022-05-04
    To 2022-05-11

Total Issues
    19 (19 resolved)

Scope

We audited commit `a95f0f6499be396c0a245f0183009b98cc72f123` of the `PaymentsERC20Public` repository.

In scope were the following contracts:

```
contracts
├── ERC20
│   ├── ERC20Token.sol
├── Payments
```

```
|    ├── IPaymentsERC20.sol
|    ├── Operators.sol
|    ├── PaymentsERC20.sol
├── Migrations.sol
```

System Overview

The Freeverse platform is a layer 2 for minting, evolving, and trading mutable NFTs with on-chain certifiable attributes ('Living Assets'), as well as traditional immutable NFTs.

The audited system deals with the ERC20 token payments used for the trade of the assets. The `PaymentsERC20` contract defines the type of `ERC20` tokens used in the trade along with a `paymentWindow` in which the payment against an asset needs to be completed.

The `buyer` has to transfer an amount of `ERC20` tokens to the `PaymentsERC20` contract for the purchase of an asset. The terms of the purchase are agreed upon by the `buyer` and `seller` outside of this audited system in presence of a special user called the `Operator`. The system assumes that the `Operator` is a trusted third-party that would ensure that both the `buyer` and the `seller` would agree to the terms of the asset purchase.

The transfer of the `ERC20` tokens from the buyer's account can be triggered by either the `Operator` by providing the buyer's signature on the payment details, or by the `buyer` by providing the `Operator's` signature on the payment details. If a `buyer` has a balance of `ERC20` tokens deposited in this contract at the start of a new payment, the contract reuses the existing tokens, and only transfers from the `buyer's` account if additional tokens are required.

Once the `ERC20` tokens are transferred to this contract, they remain locked until the `Operator` confirms the asset has been transferred from the `seller` to the `buyer`. The asset transfer itself is outside the scope of this audit.

If the asset transfer was successful, the amount of `ERC20` tokens required for the asset purchase minus the fee is credited to the `seller's` balance. If the asset transfer failed, then a refund is initiated to the local balance of the `buyer`.

The `PaymentsERC20` contract does not transfer funds to the user's account automatically. A user can call the `withdraw` function to withdraw all of their tokens from the system.

Privileged Roles

Owner

The account that deploys this system has the privileged role of the `owner`. The ownership can be renounced or transferred.

The `owner` is responsible for:

- Setting the default fee collector
- Setting a universe fee collector
- Removing a universe fee collector
- Setting the default operator
- Setting a universe operator
- Removing a universe operator
- Setting a payment window
- Setting if seller registration is required

Operator

An `operator` is an account which is responsible for ensuring that the buyer and seller are meeting the agreed terms of an asset purchase.

The operator provides their signature on the `PaymentInput` which is required to transfer tokens from the buyer's account to the system. The operator can trigger the transfer themselves with the buyer's permission.

The operator also provides their signature on the `AssetTransferResult` which indicates whether the asset transfer from the seller to the buyer was successful or not.

The role of the operator can be either `default` which acts an operator of the entire system, or specific to a universe. At the time of deployment, the `owner` is assigned as the default operator.

***Update:** In the current design of the payment system, the operator role wields a significant amount of power. A buyer cannot initiate a payment without the approval of an operator, and a seller must rely on an operator to vouch for the transfer of their asset. A corrupt operator has the ability to steal funds or assets from the system; the detailed attack is described in C01. However, after discussing with the Freeverse team, we have concluded that operators are explicitly trusted to be good actors in the system.*

FeesCollector

Fee collector is the address where a fee is credited when a payment is successfully completed. The fee is paid in `ERC20` tokens defined in the `PaymentsERC20` contract. The system does not transfer the fee to the fee collector's address automatically, but adds it to their internal balance.

The role of the fee collector can be either `default` which acts a fee collector of the entire system, or specific to a universe. The system allows only one default fee collector. If a universe has a fee collector defined, the fee goes to the universe fee collector and not the default fee collector. At the time of deployment, the `owner` is assigned as the default fee collector.

Findings

Here we present our findings.

Critical Severity

Operators are implicitly trusted

In the current design of the payment system, the operator role wields a significant amount of power. A buyer cannot initiate a payment without the approval of an operator, and a seller must rely on an operator to vouch for the transfer of their asset. Operators are implicitly trusted to be good actors in the system. However, there are four possible scenarios where operators could behave maliciously:

buyer an operator could defraud a seller by obtaining a refund even though the asset was received. After the seller's asset is transferred, the operator could either create a signed `AssetTransferResult` with the `wasSuccessful` field set to `false`, or not sign an `AssetTransferResult` at all. The operator could then call `refund` or `refundAndWithdraw` after the payment window has expired, and obtain their full payment amount even though the seller's item was successfully transferred.

- *Operator as a seller*: The contracts do not restrict an operator from also being a seller in a transaction they are responsible for monitoring. In this case, an operator could register as a seller and post an item for sale. After a buyer initiates payment, the operator could craft an `AssetTransferResult` with the `wasSuccessful` field set to `true`, but then choose not to transfer the asset. The operator could then call `finalize` or `finalizeAndWithdraw` with the signed `AssetTransferResult`, prior to the payment window expiring, thus receiving payment for an asset that was never delivered.
- *Operator colluding with a buyer*: Similar to the case where the operator and the buyer are the same, an operator could work together with a buyer in an attempt to obtain a refund for a transferred asset. As before, the operator could create a signed `AssetTransferResult` with `wasSuccessful` set to `false`, even though the asset was transferred.
- *Operator colluding with a seller*: Similar to the case where the operator and the seller are the same, an operator could work in conjunction with a seller who owns an item of value. As before, the operator would create a signed `AssetTransferResult` with `wasSuccessful` set to `true`, even though the seller did not transfer the asset.

An operator should always be an observer of transactions and never participate as a buyer or seller. To prevent misuse of the power granted to an operator, consider adding checks which ensure that the `buyer` and `seller` addresses in the `PaymentInput` struct can never be the same as the `operator` address for the same payment transaction. Also consider revisiting the trust assumptions that allow an operator entity to independently vouch for the success or failure of an asset transfer via the `AssetTransferResult` mechanism, and how the system can guarantee that an asset transfer result is always produced.

cannot be a buyer or a seller, this is fixed as of commit `714e99e` of PR #2.

Medium Severity

Potential loss of access to funds if payment window is incorrectly set

The `_paymentWindow` variable within the `PaymentsERC20` contract stores the maximum amount of time allowed for an asset transfer to be completed once a payment starts. The default value is 30 days and during this time, the buyer's funds remain locked. If the buyer tries to obtain a refund before the payment window has expired, the refund request will not be accepted.

The `PaymentsERC20` contract has a `setPaymentWindow` function that allows the contract owner to adjust the payment window for future payments. This function does not impose a limit on the chosen window size, making it possible to set the value so high that obtaining a refund from an asset transfer that failed to complete would effectively be impossible.

To prevent potential loss of access to funds, within the `setPaymentWindow` function, consider adding an upper limit on the payment window duration that the function will accept.

Update: *Fixed as of commit `a476fa1` of PR #3.*

Low Severity

Duplicate code

Duplication of code is error prone because the repeated implementations can get out of sync as the codebase evolves, potentially leading to unexpected behavior. There two instances of code duplication in the `PaymentsERC20` contract:

- The majority of the code in the `pay` and `relayedPay` functions in `PaymentsERC20` is duplicated. Once the initial checks are completed in each case, all of the remaining steps are identical. Lines 102-123 in `relayedPay` are nearly identical to lines 141-162 in `pay`, with the only difference being line 106 vs line 145, which are functionally equivalent in setting the operator. Consider creating a shared private function for this duplicate code.

external
contract, `PaymentERC20` implements `allowance` and `erc20BalanceOf` view functions that make respective calls to the `allowance` and `balanceOf` functions of the `IERC20` interface. However, when `maxFundsAvailable` performs the allowance and balance queries, it calls the provided `allowance` function, but not the provided `balanceOf` function. In the latter case, `IERC20.balanceOf()` is called directly, duplicating the code in the `erc20BalanceOf` function. Consider using the `erc20BalanceOf` function to obtain the external token balance `erc20Balance`.

***Update:** Fixed as of commit `645f3f4` of PR #1.*

Inconsistent and lack of indexed event parameters

In the `IPaymentsERC20` contract, the `Paid` event has an unindexed `paymentId` parameter, even though `paymentId` is indexed in the `BuyerRefunded` and `Payin` events.

In `FeesCollectors.sol`, the `DefaultFeesCollector` and `UniversalFeesCollector` events do not apply the `indexed` keyword to the `feesCollector` or `universeId` parameters.

In `Operators.sol`, the `DefaultOperator` and `UniverseOperator` events do not apply the `indexed` keyword to the `operator` or `universeId` parameters.

Indexed parameters are useful for quick offchain indexing of logs. Consider indexing applicable event parameters to support the searching and filtering abilities of offchain services.

***Update:** Fixed as of commit `745c035` of PR #4.*

Missing docstrings

The following functions in the codebase lack documentation:

- All the functions, events, and state variables in the `FeesCollectors` contract are missing docstrings the `IEIP712Verifier` interface have no comments explaining any of the fields

- None of the events in the `IPaymentsERC20` interface have any documentation
- There are no docstrings in the `MyToken` contract to explain the contract's purpose

This lack of documentation hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned, and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Fixed as of commit `d311775` of PR #5. However, there are a few typographical errors in the newly added docstrings: – In the docstring above the `MyToken` contract, "implementation" is misspelled as "implentation" – In the docstring above the `AssetTransferResult` struct, "ASSET_TRANSFERRING" is misspelled as "ASSET_TRANSFERING"; PR #26 moves this line to `ISignableStructs.sol`*

No check that seller is already registered

The `registerAsSeller` function in the `PaymentsERC20` contract emits a `NewSeller` event when a user calls it. There is no check in `registerAsSeller` to determine if a user address is already registered, so this function will emit a `NewSeller` event every time it is called. Depending on how the event log is used, this may lead to incorrect behavior by external observers of the contract if a user calls the function more than once.

Consider checking the existing bool value of `_isRegisteredSeller` when `registerAsSeller` is called, and reverting if the user is already registered.

Update: *Fixed as of commit `6c7a698` of PR #6.*

Some suggestions are:

- `States` enum in the `IPaymentsERC20` interface should be `State` enum
- In the `EIP712Verifier`, the variable name `inp` is confusing as it indicates both `PaymentInput` and `AssetTransferResult` structs. Consider having different names for different parameters.
- Throughout the `PaymentsERC20` contract, the local variable name for `Payment` struct is `p`.

Consider having a meaningful naming convention for variables instead of using single letters such as `p` or abbreviated words such as `inp`.

Update: *Fixed as of commit `4c2dc66` of PR #7.*

Registration process is not clearly documented

The `PaymentsERC20` contract allows a seller to register themselves by calling the `registerAsSeller` function. However, the system allows purchase of assets from unregistered sellers if the `_isSellerRegistrationRequired` flag is not set. In the scenario where `_isSellerRegistrationRequired` is set to false, any calls to `registerAsSeller` would be meaningless and would result in seller paying for the gas used for making this unnecessary transaction.

The codebase fails to document the meaning of “registration”, the advantage of being a registered seller, and why is it okay to disable this process.

Consider adding proper documentation as to why this process is required in the system and the scenarios in which it can be changed or disabled.

Update: *Fixed as of commit `65269d9` of PR #8. However, there is a typographical error in the newly added docstring. In the `README` and docstring in the `PaymentsERC20` contract, “executed” is misspelled as “exectuted”.*

Notes & Additional Information

time and has no mechanism to change the value afterward. This variable can be marked as immutable to save on gas costs when reading the value.

Consider marking this variable `immutable` in order to save users gas fees.

***Update:** Fixed as of commit `0f8028c` of PR #10.*

Contract file and contract name do not match

The `ERC20Token.sol` file, which is used only for test purposes and is not deployed, contains the `MyToken` contract. To follow common coding conventions, consider renaming `ERC20Token.sol` to `MyToken.sol` so that the contract name and the filename are the same.

***Update:** Fixed as of commit `2172356` of PR #11.*

Use of draft EIP code

The `EIP712Verifier` contract uses OpenZeppelin's implementation of the draft EIP712 standard for hashing and signing of struct data. Until the EIP712 proposal is finalized, it is possible that the standard will change, which could require updates to the `EIP712Verifier` contract. Because the `PaymentsERC20` contract inherits from `EIP712Verifier`, the end result could require a redeployment of the `PaymentsERC20` contract which stores payment data and user funds.

Consider refactoring the code to allow `EIP712Verifier` to be deployed as a separate contract that could be upgraded in isolation in the event of a change to the draft EIP712 standard.

***Update:** Fixed as of commit `d08c9f0` of PR #26.*

Imports not grouped together

The following contracts have a docstring inserted in between the import statements:

- `EIP712Verifier.sol` – import on line 16 is separated from the imports on lines 4-5
- `IPaymentsERC20.sol` – import on line 41 is separated from the imports on lines 4-6

statements together.

Update: *Fixed as of commit `2bba4d4` of PR #13.*

Misleading documentation

For clarity, consider revising the following comments which may cause confusion:

In `IEIP712Verifier.sol`, lines 10-11 state:

```
 * @dev This contract just defines the structure of a Payment Inpu
*  and exposes a verify function, using the EIP712 code by OpenZep
```

However, the contract defines two structures and two verify functions, not just `PaymentInput` and `verifyPayment` as indicated by the docstring.

In `PaymentsERC20.sol`, lines 18-20:

```
 * To start a payment, the signatures of both the buyer and the Ope
* - in the 'relayedPay' method, the Operator is the msg.sender, an
* - in the 'pay' method, the buyer is the msg.sender, and the oper
```

This docstring could be misinterpreted to mean that both `buyerSig` and `operatorSig` are required to start a payment, i.e. that both functions must be called. If `buyerSig` is provided, the operator's signature is provided by signing the `relayedPay` transaction, and similarly if `operatorSig` is provided, the buyer's signature is provided by signing the `pay` transaction. This comment is also repeated in `IPaymentsERC20.sol` and `README.md`.

Update: *Fixed as of commit `f580cf5` of PR #14.*

Mismatched parameter name between interface and contract

In the `EIP721Verifier` contract, the `verifyAssetTransferResult` function declaration has the parameter `AssetTransferResult calldata inp`, but the incorrect docstring in this case.

To avoid confusion, consider renaming these parameters so that the names match each other, and match the corresponding docstring @param tag.

Update: *Fixed as of commit `4c2dc66` of PR #7.*

Typographical errors

Consider addressing the following typographical errors:

In `IPaymentsERC20.sol`:

- line 12: “success of failure” should be “success or failure”
- line 16: “all of buyer’s received tokens” should be “all tokens received from the buyer”
- line 24: “has non-zero local balance” should be “has a non-zero local balance”
- line 28: “States Machine” should be “State Machine”
- line 34: “the a payment” should be “the payment”
- line 36: “throught” should be “throughout”
- line 47: “Payin” should be “PayIn”
- line 232: “ot check” should be “to check”

In `PaymentsERC20.sol`:

- line 12: “success of failure” should be “success or failure”
- line 16: “all of buyer’s received tokens” should be “all tokens received from the buyer”
- line 24: “has non-zero local balance” should be “has a non-zero local balance”
- line 28: “States Machine” should be “State Machine”
- line 34: “the a payment” should be “the payment”
- line 36: “throught” should be “throughout”

In `README.md`:

- line 16: “success of failure” should be “success or failure”
- line 19: “recevied” should be “received”
- line 20: “all of buyer’s received tokens” should be “all tokens received from the buyer”

line 13: "throught" should be "throughout"

***Update:** Fixed as of commit `fa5cb16` of PR #15.*

Unnecessary use of uint16

The `PaymentInput` struct defined in the `IEIP712Verifier` contract contains a `feeBPS` variable whose type is `uint16`. This value is used in the `_finalizeSuccess` function to determine the transaction fee which is subtracted from the seller's gross sale proceeds. The `feeBPS` value is passed by `_finalizeSuccess` to the `computeFeeAmount` function, which performs the actual fee calculation.

Although `feeBPS` is a `uint16` value, the `computeFeeAmount` function expects `feeBPS` to be a `uint256` value, so an explicit cast to `uint256` is performed when the function is called by `_finalizeSuccess`. Since `feeBPS` is converted to a `uint256` type at the point of use, and the variable is not being packed in the `PaymentInput` struct, it is unnecessary to store the value in a `uint16` type.

Consider changing the `feeBPS` variable type from `uint16` to `uint256`.

***Update:** Fixed as of commit `3e21d35` of PR #17.*

Unnecessary inheritance

The `FeesCollectors` contract inherits from the `Operators` contract, effectively concatenating the two independent contracts. These contracts do not share any state or functions that require the use of inheritance.

To favor clarity in the code hierarchy, consider removing the inheritance relationship in this case, allowing `FeesCollectors` to independently inherit from `Ownable`, and `PaymentsERC20` to inherit from both `FeesCollectors` and `Operators` rather than just `FeesCollectors`.

***Update:** Fixed as of commit `cf5e0ea` of PR #18.*

Unused imports

- `IERC20.sol`
- `FeesCollectors.sol`
- `EIP712Verifier.sol`

In the `IEIP712Verifier` contract:

- `draft-EIP712.sol`
- `ECDSA.sol`

Update: *Fixed as of commit `b633fdc` of PR #19.*

No functionality to withdraw portion of funds

The current system does not allow the users to take out a portion of their funds. The funds can be withdrawn from the `PaymentsERC20` contract by calling either the `finalizeAndWithdraw`, `refundAndWithdraw` or the `withdraw` functions which in-turn calls the private `_withdraw` function. This `_withdraw` function checks the balance of the caller and transfers the entire balance to the caller's address.

Although there are no security concerns with withdrawing the entire balance, it might be advisable to have an option of withdrawing a portion of funds to provide a better user experience. For example, if a user has made a profit on selling an asset, they might want to withdraw only the profitable amount and reuse the remaining funds for future trades. In the current scenario, the user would be forced to take out all of their balance and then transfer funds back to the system in order to engage with future trades, and subsequently pay for the gas for making the additional transaction.

Consider adding functionality that allows the users to withdraw a portion of their funds.

Update: *Fixed as of commit `20af988` of PR #23.*

Conclusions

OpenZeppelin

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

OpenZeppelin