2024-07-04 Decentraland offchain marketplace

Date: 2024-07-04
Author:
Facundo Spagnuolo
Commit:
2208f3bc61913901a52237e0cc62bf6fc0570e05
Amendment:
8950b0941af42140d22c3d2ef344920c0b07dde3

Introduction

This report presents the results of the security analysis conducted on the smart contracts developed for implementing an off-chain marketplace for Decentraland. The audited commit is 2208f3bc61913901a52237e0cc62bf6fc0570e05.

The repository contains multiple smart contracts to implement two marketplaces: one for Ethereum and one for Polygon, enabling users to perform different types of trades using EIP712 signatures.

Assumptions

A1. The owner is considered safe

All owner addresses are assumed to be well-protected and sufficiently decentralized.

A2. Ethereum will not support coupons for now

Currently, there are no plans to support coupons on the Ethereum mainnet.

A3. ETH is not supported for trades

Trades will only support ERC721s or ERC20s, excluding native tokens.

Findings

Critical

C1. Wrong usage of MANA/USD rate

> Note — This issue was discovered by the Decentraland development team

The `AggregatorHelper` contract exposes an internal function called `_updateAssetWithConvertedMANAPrice` that is used by both marketplaces `DecentralandMarketplaceEthereum` and `DecentralandMarketplacePolygon` in order to compute the amount that has to be paid in MANA using the current rate MANA/USD provided by ChainLink (see L50). However, the amount in USD is being multiplied by the rate when it should be divided by it. Consider updating the formula accordingly.

Update: Addressed at 78488fea753499847d4f1101d7c22a8a0b923854.

High

H1. Ethereum pays fees for ERC20 trades but Polygon does not

The `DecentralandMarketplaceEthereum` contract defines an internal function called `_transferERC20` that is used every time an asset involved in the trade is an ERC20. This function always charges a fee based on the amount being transferred (see L124). However, the marketplace defined for Polygon `DecentralandMarketplacePolygon` has a more complex fee structure:

- Decentraland NFTs collections will pay royalties
- Non-Decentraland NFT collections will pay fees
- Mint new collection NFTs will pay fees

Note the case where ERC20s are being traded is not covered above. Consider either charging fees for ERC20 trades in Polygon's marketplace or disabling this scenario in the one for Ethereum to stay consistent and avoid allowing users to bypass these fees if necessary.

Update: Addressed at 35ea625dcab6117d2f3be060406a132337ada416.

H2. Coupon discounts do not contemplate the USD-MANA asset type

The `CollectionDiscountCoupon` defines a function called `applyCoupon` that receives the original trade and changes the assets' value in order to impact the discount offered by the coupon being used (see L43-94). This function validates that the asset being traded must be an ERC20, otherwise it reverts (see L78). This validation only considers the case where the asset type is equal to `ASSET_TYPE_ERC20`, but this is not the only trade possible using ERC20 tokens. Consider also bearing in mind the case of trades that make use of the USD-pegged-MANA asset type which is `ASSET_TYPE_USD_PEGGED_MANA`.

Update: Addressed at 667cbe8260cb6523a0e32da5009caf6faa7e60a6.

Medium

M1. Fingerprints are cast to bytes32

The `DecentralandMarketplaceEthereum` contract defines a private function called `_transferERC721` used to transfer ERC721 assets to the beneficiary. In this function, the `extra` property provided for each `Asset` is used as the expected fingerprint in case the asset being transferred does support fingerprint verification (see L156-160). To do that, it calls `verifyFingerprint` to the given `asset` while providing the `extra` field cast to `bytes32` (see L156). The problem is that the ERC721 interface for fingerprint verification relies on a `bytes` value instead of `bytes32` (see L8). This could cause false negatives while invalidating trades that are actually valid. Consider avoiding the cast to `bytes32` and directly forward the `asset`'s `extra` property as it is to the ERC721 contract.

Update: Addressed at b7ff08ce59d04b2d0c64d1e1396c19519c3c6f41.

M2. Fingerprints are not supported for Polygon

The `DecentralandMarketplaceEthereum` contract defines a private function called `_transferERC721` used to transfer ERC721 assets to the beneficiary. In this function, the `extra` property provided for each `Asset` is used as the expected fingerprint in case the asset being transferred does support fingerprint verification (see L156-160). However, the `DecentralandMarketplacePolygon` contract does not rely on any fingerprint check when transferring ERC721 assets. Consider supporting this interface usage here as well to stay consistent and compliant with the expected standard.

Update: The following answer was stated by Decentraland — "This is because there are no Decentraland assets in polygon that need to verify a fingerprint like with Estates in Ethereum."

M3. Trades will not be able to be tracked easily off-chain

The `Marketplace` contract defines an event called `Traded` with the following signature: `event Traded(address indexed _caller, bytes32 indexed _signature)` (see L18). This event is emitted every time a new trade is executed. The problem is that the event only logs the caller of the transaction to the `Marketplace` contract and the hash of the signature provided to execute the trade. Note this is not enough information to reconstruct the trade off-chain in case someone needs to track these. Having the signature hashed will force trackers to decode the raw input of a transaction to detect what happened. Moreover, there are further accounting logic related to fee payment that will be missed forcing trackers to recalculate it off-chain too. Consider improving the emitted events to ensure better tracking off-chain.

Update: The following answer was stated by Decentraland — "They hashed signature is intended to be used as id, all of the Trade data will be stored on a backend and matched with it."

M4. Fee collector precision may not be enough

The `FeeCollector` contract defines a `feeRate` value to denote the rate that could be charged to different trade operations (see L14). The `feeRate` can be changed using the internal function `_updateFeeRate` which is called from the inheriting contracts to allow them to define their own authorization logic (see L36-40). However, there are no checks at all performed along these calls. Moreover, the `feeRate` is expected to be up-scaled using 4 decimals to avoid rounding errors (see L13). However, this precision without a valid rate range may not be enough.

Imagine a scenario where the `feeRate` is set to 0.01% and the amount of tokens being sent is 1,000. Then, the fee computation would be:

```
uint256 fee = 1000 * 100 / 1_000_000 // = 0.1
```

Solidity does not handle fractions and will automatically round down to the nearest integer, which in this case is 0. Thus, despite there being a nominal fee rate, the actual fee collected would be zero due to the rounding. Consider requiring a minimum fee rate that makes sense to avoid these scenarios.

Update: The following answer was stated by Decentraland — "The values we intend to use should not have an issue with this."

M5. Royalty rate precision may not be enough

The `DecentralandMarketplacePolygon` contract defines a `royaltiesRate` value (see L42). to denote the rate that could be charged to different trade operations. The `royaltiesRate` can be changed using the internal function `_updateRoyaltiesRate` (see L289-293). However, there are no checks at all performed along these calls. Moreover, the `royaltiesRate` is expected to be up-scaled using 4 decimals to avoid rounding errors. However, this precision without a valid rate range may not be enough (see L236).

Imagine a scenario where the `royaltiesRate` is set to 0.01% and the amount of tokens being sent is 1,000. Then, the fee computation would be:

```
uint256 royaltyFees = 1000 * 100 / 1_000_000 // = 0.1
```

Solidity does not handle fractions and will automatically round down to the nearest integer, which in this case is 0. Thus, despite there being a nominal fee rate, the actual fee collected would be zero due to the rounding. Consider requiring a minimum fee rate that makes sense to avoid these scenarios.

Update: The following answer was stated by Decentraland — "The values we intend to use should not have an issue with this."

Low

L1. `Trade` object is modified both internally and externally

Among most of the marketplace-related smart contracts, there is a common pattern that relies on the modification of the `Trade` object which is given initially by the caller. Some of these examples are:

- The `DecentralandMarketplacePolygon` overrides the `_modifyTrade` method defined initially in the `Marketplace` contract, in order to modify the assets being sent and received to consider the potential fees and royalties information (see L116-L141).
- The `CouponManager` defines a function called `applyCoupon` that receives the original trade, forwards it to the coupon contract being used, and forwards back the trade returned by the coupon contract (see L77-109).
- The `CollectionDiscountCoupon` defines a function called `applyCoupon` that receives the original trade and changes the assets' value in order to impact the discount offered by the coupon being used (see L43-94).

Even though these scenarios have been analyzed to make sure there is no mistaken manipulation of the `Trade` object, this is not a recommended pattern. Note you're relying completely on functions being called in a specific order, if some of these are re-arranged, accounting could be messed up. Also, the protocol does not guarantee any new components like coupons may not manipulate the trade information in a bad way. Any future changes or dependency updates must be reviewed critically bearing in mind the whole execution context, it cannot be analyzed in an isolated manner.

A simple alternative could be to at least define different internal struct objects, even if they follow the same data structure, it could help at compile time to make sure what is expected from and out of each contract.

Update: The following answer was stated by Decentraland — "This is by design."

L2. Unclear usage of `Asset`'s `extra` property

The `MarketplaceTypes` contract defines the `Asset` struct (see L16-22) that has a property named `extra` intended to store any additional information (see L15). This property has different important usages that can be found depending on the marketplace contract.

On one hand, the `DecentralandMarketplaceEthereum` uses the `extra` field to read the expected fingerprint for ERC721 trades. On the other hand, the `DecentralandMarketplacePolygon` uses the `extra` property to store local fees and royalties information that must be paid later one (see L191). As you can see, in the first example this field is used to store information provided by the user, while in the second case, this property is used to as a cache by the marketplace itself to perform some accounting computation. This pattern is a bit inconsistent and as mentioned in the previous finding, depending on the order in which certain memory slots are updated might be a strong coupling.

Consider making explicitly clear what's the purpose of this field and avoid using it for local computation in case it is expected to be populated by the user.

Update: Addressed at 9392e7c12effe028bd29e9fcab64fa7ce4ae8394.

L3. Polygon marketplace overrides trades types

The `DecentralandMarketplacePolygonAssetTypes` defines two custom trade types `ASSET_TYPE_COLLECTION_ITEM` and `ASSET_TYPE_ERC20_WITH_FEES` (see L8-9). This contract also defines a method called `_updateERC20sWithFees` where the `Trade` object is updated using the fees and royalties computed, but it also overrides the `assetType` of the `Trade` to `ASSET_TYPE_ERC20_WITH_FEES` in case the one provided by the user is `ASSET_TYPE_ERC20` (see L190).

Similar to what was mentioned above, the idea of modifying the input given by the user in order to affect the way specific computation should behave later on seems a bit sketchy. Consider avoiding to override the input provided by the user as much as possible.

Update: Addressed at 7ab409959a8f8b6150200e2e3ea5293a1415cfe9.

L4. Overridden unnecessary asset type

The `DecentralandMarketplacePolygonAssetTypes` defines two custom trade types `ASSET_TYPE_COLLECTION_ITEM` and `ASSET_TYPE_ERC20_WITH_FEES` (see L8-9). This contract also defines a method called `_updateERC20sWithFees` where the `Trade` object is updated using the fees and royalties computed, but it also overrides the `assetType` of the `Trade` to `ASSET_TYPE_ERC20_WITH_FEES` in case the one provided by the user is `ASSET_TYPE_ERC20` (see L190). However, this overridden type does not seem to affect much, the assigned value is simply changed and it seems to be used simply to make sure fees were actually computed. Consider removing this new internal type and work always with the input provided by the user.

Update: Addressed at 7ab409959a8f8b6150200e2e3ea5293a1415cfe9.

L5. Fee collector can be set to zero

The `FeeCollector` contract defines a `feeCollector` address where all the fees charged to the processed trades will be deposited (see L12). This address can be changed using the internal function `_updateFeeCollector` which is called from the inheriting contracts to allow them to define their own authorization logic (see L27-31). However, there are no checks at all performed along these calls. Consider at least requiring the address not to be zero to avoid setting an undesired value that could result in a loss of fees.

Update: The following answer was stated by Decentraland — "We don't think owner controlled values need to have checks."

L6. Fee can be set above 100%

The `FeeCollector` contract defines a `feeRate` value to denote the rate that could be charged to different trade operations (see L14). The `feeRate` can be changed using the internal function `_updateFeeRate` which is called from the inheriting contracts to allow them to define their own authorization logic (see L36-40). However, there are no checks at all performed along these calls. Therefore, the `feeRate` can be set above 100%, which will cause the fee computation to fail undesirably when the trade is executed. Consider defining a maximum value to validate the `feeRate`.

Update: The following answer was stated by Decentraland — "We don't think owner controlled values need to have checks."

L7. Royalty manager can be set to zero

The `DecentralandMarketplacePolygon` contract defines a `royaltiesManager` address where all the royalty fees charged to the processed trades will be deposited (see L39). This address can be changed using the internal function `_updateRoyaltiesManager` which is called from the inheriting contracts to allow them to define their own authorization logic (see L281-285). However, there are no checks at all performed along these calls. Consider at least requiring the address not to be zero to avoid setting an undesired value that could result in a loss of royalty fees.

Update: The following answer was stated by Decentraland — "We don't think owner controlled values need to have checks."

L8. Royalty rate can be set above 100%

The `DecentralandMarketplacePolygon` contract defines a `royaltiesRate` value to denote the rate that could be charged to different trade operations (see L42). The `royaltiesRate` can be changed using the internal function `_updateRoyaltiesRate` (see L289-L293). However, there are no checks at all performed along these calls. Moreover, the `royaltiesRate` can be set above 100%, which will cause the royalty fee computation to fail undesirably when the trade is executed. Consider defining a maximum value to validate the `royaltiesRate`.

Update: The following answer was stated by Decentraland — "We don't think owner controlled values need to have checks."

L9. Royalty rate and fee collector rate may not add up

Both the `FeeCollector` and the `DecentralandMarketplacePolygon` contracts defines a `feeRate` and `royaltiesRate` respectively to denote the rate that could be charged to different trade operations. The `feeRate` can be changed using the internal function `_updateFeeRate` which is called from the inheriting contracts to allow them to define their own authorization logic, while the `royaltiesRate` can be changed using the internal function `_updateRoyaltiesRate`. However, there are no checks at all performed in any of these functions. The problem is that the sum of both rates can be above 100%, which will cause the general fee computation to fail undesirably when the trade is executed. Consider adding an internal check to make sure the total fee rate does not exceed 100%.

Update: The following answer was stated by Decentraland — "We don't think owner controlled values need to have checks."

L10. Check abi decodes length if considered needed

Many different contracts use `abi.decode` to parse arbitrary data into different expected data types. However, the data being parsed is not checked which causes a decode failure in case the expected type does not match the given data. Some of these examples are:

- The `CollectionDiscountCoupon` contract decodes the `Coupon`'s `data` property into a `CollectionDiscountCouponData` struct (see L48).
- The `CollectionDiscountCoupon` contract decodes the `Coupon`'s `callerData` property into a `CollectionDiscountCouponCallerData` struct (see L49).
- The `Verifications` contract decodes the external check responses in different expected values (see L124-135).

Consider implementing type checks before decoding these values in case you consider it important not to fail undesirably.

Update: The following answer was stated by Decentraland — “It is ok for the transaction to fail if the decode fails.”

L11. Make marketplace functions that must be overridden abstract

The `Marketplace` contract is an abstract contract that defines different `virtual` functions that should be overridden by their inherited implementations, these are `_modifyTrade` (see L133) and `_transferAssets` (see L138). However, instead of providing a default implementation from the abstract contract and commenting that these must be overridden, consider making them `abstract` which will force the developer to do this even though if they provide an empty implementation for them. Otherwise, it would be easy to miss the case where you need to consider a particular scenario when developing a custom marketplace.

Update: Addressed at deb4584e628dd1881dcabdc4fe6b98b871830e45.

L12. Validate chain id in constructors for Marketplace contracts

The `Marketplace` contract is an abstract contract inherited by the `DecentralandMarketplacePolygon` and `DecentralandMarketplaceEthereum` contracts. Based on their names these contracts are seem to be deployed to Polygon and Ethereum respectively. Consider checking the `block.chainid` in the constructors to make sure these are not deployed to an undesired network.

Update: The following answer was stated by Decentraland — “Given that the contract is deployed by Decentraland. Mixing up the chain should not be an issue.”

L13. Call constructors in order

Both the `DecentralandMarketplacePolygon` and `DecentralandMarketplaceEthereum` contracts inherit multiple smart contracts. Solidity uses a linearization algorithm to translate multiple inheritance into a single chain path. Ideally, upper constructors should be called in order, instantiating the constructor of the first contract in the inheritance chain in the first place.

Therefore, both contracts should call the upper contracts’ constructor in the following order: `FeeCollector`, `EIP712`, `Owanble`, and `MarketplaceWithCouponManager`. Consider adjusting this to make sure every contract has properly initialized whatever information they might be relying on based on their upper contracts.

Update: Addressed at 0e56f57a88a4c619b92b8578cfa3626b70ceb400

L14. Only a few errors use parameters

Along the entire codebase, only a few custom errors declare specific parameters.

- `Signatures`:
  - `InvalidSignature`

- `Verifications`:
  - `UsingCancelledSignature`
  - `SignatureReuse`
  - `NotEffective`
  - `InvalidContractSignatureIndex`
  - `InvalidSignerSignatureIndex`
  - `Expired`
  - `NotAllowed`
  - `ExternalChecksFailed`
- `NativeMetaTransaction`:
  - `MetaTransactionFailedWithoutReason`
- `CollectionDiscountCoupon`:
  - `InvalidSentOrProofsLength`
  - `InvalidDiscountType`
- `CouponManager`:
  - `LengthMissmatch`
- `Marketplace`:
  - `UsedTradeId`
- `DecentralandMarketplacePolygon`:
  - `NotCreator`
- `DecentralandMarketplaceEthereum`:
  - `InvalidFingerprint`

Consider adding parameters to provide context information when you consider it necessary.

Update: Acknowledged

L15. Inconsistent empty trades checks

The `Marketplace` contract defines a function called `accept` in order to submit a trade a caller is willing to accept (see L49). This function does allow submitting empty trades, more specifically, with an empty list of assets to be received and an empty list of assets to be sent. On the other hand, the `MarketplaceWithCouponManager` declares another method called `accepWithCoupon` that should be used in order to accept a trade along with a coupon (see L25), which ends up calling the submitted coupon contract through the `CouponManager`. In particular, the `CollectionDiscountCoupon` contract does validate that the submitted trade is not empty (see L51). Consider either supporting or denying empty trades to stay consistent.

Update: Addressed at 7d0b334db435562830062e6f1a9395aef95bf287

L16. Inconsistent internal transfer functions visibility

The `DecentralandMarketplaceEthereum` contract overrides the `_transferAsset` function inherited from the `Marketplace` contract. In this function, different methods are called based on the asset type being transacted, in case an ERC20 is requested the `_transferERC20` method is called, in case a MANA-pegged-USD is requested the `_transferUsdPeggedMana` method is called, and in case an ERC721 is requested the `_transferERC721` method is called. The first one is declared using an `internal` visibility modifier (see L122), and the other twos are declared using a `private` visibility modifier (see L133 and L152 respectively). Based on how the general codebase is written, usually local contract functions that are not virtual use the `private` visibility modifier. Consider declaring `_transferERC20` private to stay consistent.

Update: Addressed at 9e3224e768da6fe359bf35ff5328d45d463c6fd5

L17. Unclear trade ID purpose

The `Marketplace` contract defines a trade ID mapping to denote which trades have been executed (see L12-15). However, there is not much information on why marking a trade as executed in this particular mapping is required. Consider providing enough information to denote what's the purpose of this data structure.

Update: Addressed at 94f37dc32a49321af14d4889bb945ea6b3183c26

L18. Trade ID being changed when using USD-MANA asset type

The `AggregatorHelper` contract is mainly used to interface with ChainLink-compliant oracles to fetch rates on-chain. This contract defines a function called `_updateAssetWithConvertedMANAPrice` that is used in order to convert a trade value from USD into MANA (see L47-L50). These properties are the ones used in order to build the trade ID too (see L71). Even though this is not currently a problem since the trade ID is computed and stored before the `AggregatorHelper` is called, consider making a statement in the codebase to denote the `getTradeId` function cannot be trusted at a certain point when relying on this asset type.

Update: Acknowledged

L19. Assuming aggregators with 18 decimals or less

The `AggregatorHelper` contract is mainly used to interface with ChainLink-compliant oracles to fetch rates on-chain. This contract defines a function called `_getRateFromAggregator` that is used in order to fetch information from an on-chain aggregator and convert the result using 18 decimals. However, this function only considers the case where the decimals used by the aggregator are 18 or less (see L35). Consider tweaking this function to contemplate higher precision results to avoid having the query revert undesirably.

Update: Acknowledged

L20. Assumed USD values using 18 decimals

The `AggregatorHelper` contract is mainly used to interface with ChainLink-compliant oracles to fetch rates on-chain. This contract defines a function called `_updateAssetWithConvertedMANAPrice` that is used in order to convert a trade value from USD into MANA. However, this function assumes that the value input by the user is expressed using 18 decimals (see L50). This is not wrong, but it's a strong and important assumption, consider making this clear in the codebase or in the documentation.

Update: Addressed at fa63295470b03fd145f5ac814196494abb9badf1

Informational

I1. Fix pragma version

All the contracts in the codebase support being compiled with any compiler version above 0.8.20. Consider fixing the Solidity pragma version in case you consider mandatory to use a specific compiler version.

Update: Addressed at 367060a6dd5c087bb5b50d279d67b5a82d315dcd

I2. Use errors instead of revert reasons

The `NativeMetaTransaction` contract defines a function called `executeMetaTransaction` in order to execute transactions on behalf of someone else. This function throws a revert reason in case the signature provided is invalid (see L61). However, this is not the adopted approach compared to the rest of the codebase. Consider using a Solidity error instead of a reason string.

Update: Addressed at 50459b910c24489726b875d61ff06d6cbcf0241a

I3. Follow the same idea when computing type hashes

Some of the contracts of the codebase are inherited from the EIP712 contract implementation. Therefore, all these have to declare the different type-hashes they support to verify signatures. Some of them are hardcoded but others are computed inline. Consider following the same approach for all of them. In case you pick to hardcode these values to save gas, consider validating their expected value in the constructor of the contract.

Update: Addressed at 2d135d98acd84f6c7c19809ecd2721c4531dccb3

I4. Solve TODO comments in readme

The README file includes some TODO comments in the Signatures section (see here). Consider solving these TODO comments to provide the corresponding information.

Update: Addressed at 360371bc048128e13d8c3dcd3c4ce5f8ad7f26fb

I5. No need to use hierarchy to share structs between files

There are many contracts in the codebase that simply define data structs in order to allow sharing these among different contracts, these are `CommonTypes`, `CouponTypes`, and `MarketplaceTypes`. However, Solidity does allow importing files with structs declaration without having to define a contract in order to use inheritance to reuse these concepts.

Update: Acknowledged

I6. No license defined

Many contracts in the codebase are unlicensed with `SPDX-License-Identifier: UNLICENSED`. Consider defining the license used for each of them.

Update: Addressed at 00df5b8b5bf29f1e9627add893a1f2f90eb21c2a