2025-04-01 Decentraland Credits Manager

Date: 2025-04-01
Author:
Facundo Spagnuolo
Commit:
4fb7e3db9939fc226c22b6568338926570f249f9
Amendment:
7e2778981fee66c24cc5620e95c748b1797ff0f5

Introduction

This report presents the results of the security analysis conducted on the `CreditsManagerPolygon` smart contract developed to be integrated to Decentraland's off-chain marketplace. The audited commit is 4fb7e3db9939fc226c22b6568338926570f249f9.

Assumptions

A1. Credits are signed per user

Credits are uniquely signed and bound to the specific user address (`sender`), enforced via signature verification. Only the intended user can consume their credits.

A2. Partial credits are allowed

If a credit has remaining value, only the portion needed is consumed. The contract assumes credits can be used partially over multiple calls.

A3. Hourly limit is global across all users.

The contract assumes an hourly rate limit shared among all users and credits.

A4. Primary sales are exclusively permitted for `Marketplace` and `CollectionStore` transactions

Only transactions targeting the `Marketplace` contract with collection items (`ASSET_TYPE_COLLECTION_ITEM`) or the `CollectionStore` contract are considered valid for primary sales. All other targets or asset types are not authorized to execute primary sales.

A5. Secondary sales are exclusively permitted for `Marketplace` ERC-721 assets and `LegacyMarketplace` transactions

Only transactions involving ERC-721 assets (`ASSET_TYPE_ERC721`) within the `Marketplace` contract, and transactions processed through the `LegacyMarketplace` contract, are authorized for secondary sales. No other contracts or asset types are permitted to perform secondary sales.

A6. External call expiration applies exclusively to custom external calls

The `expiresAt` field in the `ExternalCall` struct is only enforced for custom external calls. Standard `Marketplace`, `LegacyMarketplace`, and `CollectionStore` transactions do not consider the `expiresAt` parameter during execution.

A7. Custom external calls are considered safe if explicitly allowed in mapping

The contract assumes governance properly curates safe external calls, and signatures on custom calls cannot be replayed.

A8. The contract owner (`DEFAULT_ADMIN_ROLE`) is trusted not to misuse roles

Critical roles like credit signer, pauser, user denier, and external call signer are granted at deploy time. The contract assumes role governance is secure and not misused.

Findings

Critical

None

High

H1. External custom call signature does not follow EIP-191

Custom external call signatures are recovered using `ECDSA.recover()` over a hash computed by:

```
keccak256(abi.encode(_sender, block.chainid, address(this), _args.externalCall))
```

This approach does not follow EIP-191, which recommends prefixing the signed data with the string `"\x19Ethereum Signed Message:\n32"` to prevent cross-protocol attacks.

Consider prefixing your message hashes accordingly before recovering the signer to stay compatible with standards and prevent potential misuse.

Update: Addressed at 821cf1e9265e27cbfd9f459242a50bee5288d41e

H2. Credit signatures do not follow EIP-191

Similarly to H1, credits are verified using `ECDSA.recover()` over a message hash that does not comply with EIP-191:

```
keccak256(abi.encode(_sender, block.chainid, address(this), credit))
```

This opens up the possibility of cross-protocol replay attacks and makes signature generation inconsistent with widely used tools and libraries.

Consider adopting EIP-191-compliant message construction.

Update: Addressed at 821cf1e9265e27cbfd9f459242a50bee5288d41e

H3. Malicious or unintended refunds lead to inconsistent accounting

The `CreditsManagerPolygon` contract assumes that the MANA transferred out during the external call corresponds directly to the actual consumption of funds. However, this assumption breaks if the external contract refunds MANA back to the `CreditsManagerPolygon` contract during the call.

The `_executeExternalCall` function captures the `manaTransferred` value as:

```
manaTransferred = balanceBefore - mana.balanceOf(address(this))
```

If the external call returns MANA to the contract, this refund is not subtracted from `manaTransferred`. As a result, the contract overestimates the amount spent.

This leads to multiple critical issues:

H3.1. Incorrect internal accounting

The inflated `manaTransferred` causes the following miscalculations:

- `creditedValue` is overestimated, and more credit is consumed than necessary.
- `spentValue[signatureHash]` is updated with an excessive value.
- `manaCreditedThisHour` is increased inaccurately.

This creates an inconsistent internal state: credits are underutilized, but the system marks them as partially or fully used.

H3.2. Hourly rate limit bypass

The hourly credit budget is tracked via `manaCreditedThisHour`. Since this value is based on the overestimated `manaTransferred`, users may **appear** to stay within the hourly limit while actually exceeding it. This allows malicious users or contracts to **bypass rate-limiting** using well-timed refunds.

H3.3. Reproducible example

Let's say an item costs 500 MANA and the user provides:

- A credit worth 500 MANA.
- `maxUncreditedValue = 100` MANA in case it is necessary.

At first, the contract pre-charges uncredited value as follows:

```
mana.safeTransferFrom(user, contract, 100);
```

Then, all available funds are allowed before the external call to be consumed:

```
mana.forceApprove(target, 100 + 500);
```

Let's assume the external contract refunds 100 MANA back to `CreditsManagerPolygon`. This is where the contract miscalculates `manaTransferred` by doing:

```
manaTransferred = balanceBefore - balanceAfter; // 500 - 100 = 400
```

It's important to notice the contract thinks only 400 MANA were used instead of 500. As a consequence, the contract misapplies credits:

```
creditedValue = min(credits, manaTransferred) = 400;
spentValue[creditId] += 400;
manaCreditedThisHour += 400;
```

The credit appears reusable since there is still 100 MANA available, and the hourly cap is underreported.

Fortunately, users' unused funds are correctly sent back:

```
uncredited = manaTransferred - creditedValue = 0;
mana.safeTransfer(user, 100 - 0); // User refunded 100 MANA
```

H3.4. Summary of effects

Affected Component	Expected	Actual	Result
`manaTransferred`	500	400	Underestimated
`creditedValue`	500	400	Credit underused
`manaCreditedThisHour`	+500	+400	Rate limit bypass
`spentValue[creditId]`	+500	+400	Credit appears reusable
User refund	100	100	Correct

H3.5. Recommendation

Consider tracking true MANA outflow rather than net balance delta. This ensures accurate accounting and enforcement of quotas and credit exhaustion.

Update: Acknowledged

Medium

M1. External call failures are not bubbled up

If an external call fails, the contract simply reverts with a generic `ExternalCallFailed` error, without bubbling up the underlying revert reason:

```
(bool success,) = _args.externalCall.target.call(...);
if (!success) revert ExternalCallFailed(_args.externalCall);
```

This makes debugging difficult and limits observability.

Consider using assembly to bubble up revert reasons from external calls to improve transparency and aid troubleshooting.

Update: Acknowledged

M2. `expiresAt` dates are not considered expired

The contract checks expiry timestamps using:

```
if (block.timestamp > expiresAt) {
    revert Expired();
```

```
}
```

This means that the expiration is valid until strictly greater than the expiration time, allowing usage at the exact expiration second.

Consider treating the expiration date inclusively for safer and stricter behavior.

Update: Addressed at 1cf1b1b305af50755bd1a34e23568b536defc9a1.

Low

L1. `ERC721Withdrawn` event does not index token ID

The `ERC721Withdrawn` event omits indexing the token ID:

```
event ERC721Withdrawn(address indexed _sender, address indexed _token, uint256 _tokenId, a
ddress indexed _to);
```

Since token IDs are fundamental for tracking NFT transfers, consider indexing `_tokenId` to improve observability for off-chain monitoring tools.

Update: Addressed at 7e2778981fee66c24cc5620e95c748b1797ff0f5.

Informational

I1. Extract multiple roles check to modifier

The contract manually performs role checks like:

```
if (!hasRole(DEFAULT_ADMIN_ROLE, sender) && !hasRole(PAUSER_ROLE, sender)) {
    revert Unauthorized(sender);
}
```

This logic is repeated across multiple functions.

Consider extracting these checks into dedicated modifiers to improve code clarity and reduce duplication.

Update: Acknowledged