Kreatorhood Security Review

Version 2.0

January 25, 2024

Conducted by:
George Hunter, Independent Security Researcher

Table of Contents

1 About George

George Hunter, is an independent security researcher experienced in Solidity smart contract auditing and bug hunting. Having conducted over 50 solo and team smart contract security reviews, he consistently aims to provide top-quality security auditing services. He serves as a smart contract auditor at Paladin Blockchain Security, where he has been involved in security audits for notable clients such as LayerZero, TraderJoe, SmarDex, and other leading protocols.

For security consulting, you can contact him on Twitter, Telegram, or Discord - *@GeorgeHNTR*.

2 Disclaimer

Audits are a time, resource and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	Kreatorhood
Repository	https://github.com/Kreatorhood-HD/kreatorhood-blockchain
Commit hash	9d7f4b93ae33d6bc5886f4ea908f8734d8d2520e
Resolution 1	9e336a996084e6e643ac4274664bef9b45a89f8a
Resolution 2	84c3a6548191b20506830b940a82fac59cfa1c33
Resolution 3	8a20eb77e02ab11e49a3eba304afe1038fe522d9
Methods	Manual review

Scope

contracts/*

Issues Found

Severity	Total	Resolved	Acknowledged	Failed
Critical risk	1	1	-	-
High risk	3	3	-	-
Medium risk	3	2	1	-
Low risk	7	6	1	-
Total	14	12	2	-

5 Findings

5.1 Critical risk

5.1.1 A malicious seller can steal buyers' funds from pending secondary market sales

Severity: *Critical risk*

Context: MarketplaceFacet.sol#L180-L187, NativeMetaTransactionFacet.sol#L173-L195

Description: The Kreatorhood NFT Marketplace supports 2 main types of purchasing functionalities:
- Buying an NFT from the primary market (minting an NFT).
- Buying an NFT from the secondary market (from a seller).

For the second type, users are expected to provide 2 main structs to the `buyListed` function: `SellDTO` and `Splits`. The `Splits` struct contains properties that are signed by an off-chain operator and are used to provide information regarding the payments made by the buyer to purchase a given NFT. The `SellDTO` represents the sale of the NFT and contains properties that are signed by the seller (current owner) of the given NFT for sale.

The problem is that there is no strict validation that the `SellDTO` passed to the `buyListed` function actually corresponds to the `Splits` of the buyer signed by the off-chain operator. We can see that when verifying that the `Splits` struct properties have been signed by the off-chain operator, the hash of all sell nonces is also included in the signed message payload to make the connection between the splits and the sell structs:

```
verifySplits(sellAndSplitsDTO[s].splitsInfo, keccak256(abi.encodePacked(sellNonces))
    );
```

```
function verifySplits(
    StructStorage.Splits calldata splitsInfo,
    bytes32 tradeDataHas
) internal {
    ...
    (bool valid, address signer) = IMetaTransaction(address(this))
        .tryRecoverSplitsSigner(splitsInfo, tradeDataHash);
    ...
}
```

However, the nonces are in no way linked to the actual properties of the sale, as they are just arbitrary numbers passed by the caller of the function.

Therefore, a malicious seller can build a script that detects any pending transactions for secondary market sales getting executed and front-runs them by providing the same `Splits` struct of the buyer as well as the `sellNonces` the seller is about to use but inserts different sell properties, like a mock contract address for `nftAddress`. That way, the buyer's payments will still get executed, the transaction will be successful, but the attacker (seller) will keep their NFTs.

Recommendation: One way to prevent this issue is by hashing not just the sells' nonces on line 180, but all properties of the `sellDTO`, as is done for the seller's signature. That way, both the sell and payment split will be validated by the off-chain operator, and users will not be able to choose arbitrary sell parameters for any nonce.

Resolution: Resolved. The `tryRecoverMerkleProofOrSellDataSigner` function now computes the sell nonce as the keccak256 hash of the sell properties.

5.2 High risk

5.2.1 Protocol fees can be stolen by marketplace buyers or sellers

Severity: *High risk*

Context: FundsAdminFacet.sol#L40-L42

Description: The Kreatorhood NFT Marketplace supports different payment tokens that are added using the `addAcceptedToken` function of the `MarketplaceAdminFacet` contract. One of the supported tokens is the native token for the corresponding chain.

The way payment `Splits` are distributed is by simply transferring funds from buyers to sellers. This can be seen for ERC20 tokens in the `distributeSplits` function:

```
function distributeSplits(address buyer, LibStructStorage.Splits memory splitsInfo)
    external override {
    ...
    for (uint256 i; i < paymentsLength; ) {
        IERC20(splitsInfo.token).safeTransferFrom(
            buyer,
            splitsInfo.payments[i].recipient,
            splitsInfo.payments[i].amount
        );
    ...
```

However, for native tokens like ETH, the contract expects that the buyer will first transfer the needed amount to the contract via a payable function like `buyNew` or `buyListed`, and then the same amount of native tokens will be transferred out when `distributeSplits` is called in the same transaction:

```
function distributeSplits(address buyer, LibStructStorage.Splits memory splitsInfo)
    external override {
    ...
    for (uint256 i; i < paymentsLength; ) {
        if (splitsInfo.payments[i].recipient != address(this)) {
            (success, ) = payable(splitsInfo.payments[i].recipient).call{
                value: splitsInfo.payments[i].amount
            }("");
            if (!success) revert LibStructStorage.FA_NATIVE_TRANSFER();
        }
    ...
```

The problem is that the contracts assume that a sufficient amount of native tokens has been initially sent with the call because otherwise the .call operation in `distributeSplits` would fail due to insufficient balance of the Marketplace. However, during its lifecycle, the Kreatorhood NFT Marketplace actually holds all fees collected from users in various tokens. Therefore, an attacker can simply not send any native tokens when buying an NFT (on the primary or secondary market) with native tokens and instead pay for it with the protocol fees currently staying in the Marketplace's balance.

Recommendation: Consider validating that the msg.value is exactly the same as the total amount of native tokens used when distributing payment splits.

Resolution: Resolved. The client implemented the recommended fix.

5.2.2 NFT's utility and history can be manipulated by arbitrary users upon minting

Severity: *High risk*

Context: MarketplaceFacet.sol#L109, NativeMetaTransactionFacet.sol#L118-L153

Description: The Kreatorhood NFT is an extended ERC721 NFT smart contract that also implements 2 additional standards - EIP2981 (NFT Royalty) and EIP6785 (NFT Utilities Information). The latter is written by the client (Kreatorhood) and defines standard functions and an extension of the metadata schema that outlines what a token's utility entails and how the utility may be used and/or accessed on-chain.

When minting a new Kreatorhood NFT, a `utilityUri` string is included as part of the `MintInfo` for each NFT instance. The value of this property is determined by an input array of strings named `utilityIds`, which is part of the `mintData.drops`:

```
mintInfos[nftsMinted] = IKreatorhoodERC721.MintInfo(
    mintData.drops[i].tokenUri,
    // the tokenIds array starts at 0, but we're counting editions from editionId
    getUtilityUri(mintData.drops[i].utilityIds[e - firstNewEditionId])
);
```

The problem is that, although the `mintData` is passed for verification that the contract's operator has signed all properties using EIP712 typed structured data hashing and signing, the `utilityIds` appear to not be included in the message payload signed by the operator:

```
function hashDropData(StructStorage.DropData memory dropData) public pure returns (
    bytes32) {
    return
        keccak256(
            abi.encode(
                Storage.DROP_DATA_TYPEHASH,
                dropData.dropId,
                dropData.maxEditions,
                dropData.mintEditions,
                dropData.royaltiesPercent,
                dropData.creatorAddress,
                keccak256(bytes(dropData.tokenUri)) // @audit missing ‘utilityIds‘
            )
        );
}
```

Since the `utilityIds` are actually not validated, a malicious buyer can simply choose whatever utilityId they want as the value for their `utilityUri` upon minting. Also, an adversary can front-run a purchase from a primary market transaction and change the initial utilityId of a victim buyer.

Note that the `creator` of the NFT can later manually change the utilityId (to fix it), but the one used upon minting will forever remain in the `history` of `utilityUri`s, which is a serious violation of the standard and the normal flow.

Recommendation: Include the `utilityIds` array in `DROP_DATA_TYPE` and `DROP_DATA_TYPEHASH` in `LibMetaTransactionStorage` and in the `hashDropData` method in `NativeMetaTransactionFacet`.

Resolution: Resolved. The client implemented the recommended fix.

5.2.3 Denial-of-Service attack on purchases and sales

Severity: *High risk*

Context: MarketplaceFacet.sol#L302, NativeMetaTransactionFacet.sol#L169-L201

Description: The root cause of this issue is similar to that of Finding 5.1.1 in this security review report.

The Kreatorhood NFT Marketplace implements functionality that allows NFT sellers to cancel their pending sales before they get executed by simply providing a signed `SellDTO` struct containing the sale properties and their nonces.

```
function cancelSell(StructStorage.SellDTO calldata sellDTO) external override {
    Storage.MarketplaceStorage storage data = Storage.getStorage();
    IMetaTransaction(address(this)).tryRecoverMerkleProofOrSellDataSigner(sellDTO);
    ...
```

However, similar to the issue in Finding 5.1.1 of this security review report, there is no validation that the passed nonce actually corresponds to the given `sellDTO` that should be stored and generated in the application's backend. Therefore, an adversary can simply monitor all pending transactions that will execute a purchase of an NFT on the secondary market and front-run it by calling `cancelSell` with the same `sellNonce`.

Recommendation: Consider verifying that the nonce actually belongs to the given `sellDTO` struct and its properties. This can be achieved by determining the value of the nonce using the hash of the struct. Alternatively, consider whether the off-chain operator should also sign the `sellDTO` and verify its signature as well on line 195.

Resolution: Resolved. The client implemented the recommended fix (same as for Finding 5.1.1).

5.3 Medium risk

5.3.1 Griefing attack vector can prevent users from interacting with the marketplace

Severity: *Medium risk*

Context: MarketplaceFacet.sol#L152

Description: An argument named `externalId` is passed to and used in each user-facing function of the `MarketplaceFacet` contract, except for `cancelSell`.

As mentioned in a code comment, this parameter represents a "back-end number used for processing logs efficiently." There is simple input validation that ensures that two `externalId` of the same value are never used for processing a transaction. This validation is applied in all the following functions: `buyNew`, `buyNewFiat`, `buyListed`, `buyListedFiat`, `buy`, `buyFiat`:

```
function verifyExternalId(uint256 externalId) internal {
    Storage.MarketplaceStorage storage data = Storage.getStorage();

    if (data.externalIds[externalId]) revert StructStorage.EXTERNAL_ID_USED(
        externalId);
    data.externalIds[externalId] = true;
}
```

However, used in this way, the `externalId` parameter introduces a griefing vector that allows an adversary to prevent users' transactions from executing successfully. The attack can be easily executed using a script that front-runs the victim's transaction and uses the same `externalId`. The script would simply call the `buyListed` function with an empty array and the specific `externalId` because it: 1. has no access control. 2. has all its logic inside a for-loop iterating over a user-supplied array.

Recommendation: The simplest way to mitigate the issue is by implementing a simple check for a zero-length array in `buyListed` to reduce the attack surface. A more complicated fix would require using an on-chain counter/nonce that increments each time it is used. However, the latter, although a better fix, may incur too much overhead with regard to the client's backend logic.

Resolution: Resolved. The client implemented a check for a zero-length array in the `buyListed` method.

5.3.2 Operator can steal tokens from users' balances due to leftover approvals

Severity: *Medium risk*

Context: FundsAdminFacet.sol#L51-L55

Description: The flow for paying for an NFT item bought from the primary market in the Kreatorhood NFT Marketplace is as follows: 1. The operator signs the `MintData` struct properties. 2. The operator signs a corresponding payment `splitsInfo`. 3. The buyer approves the needed amount of tokens to the marketplace. 4. Signature verification passes for the payment `splitsInfo`. 5. A user executes the `buyNew` function, which transfers the tokens from the buyer to each payment split recipient. 6. Signature verification passes for the `MintData`. 7. NFTs are minted.

As can be seen, the off-chain operator is a trusted entity that has to sign messages for users to interact with the NFT Marketplace. Although it is mostly used for verification purposes, the operator can also act maliciously in case of private key leakage. A possible attack vector in such a case would be that an operator will be able to steal all tokens from users who have approved more tokens to the marketplace than they have actually spent.

Recommendation: A potential fix for this issue is to require a signature from buyers for the execution of each action to ensure that they are aware of the properties being used within the action executed on their behalf. Another way to mitigate this risk will be by requiring that the msg.sender and `buyer` are always the same. However, there is a flow where the msg.sender can be actually the seller of the NFT, so this fix may not be practical. At the very least, advise users on the frontend side of the application to never give infinite approval to the Kreatorhood NFT Marketplace diamond proxy contract address and to revoke any leftover approvals after interacting with the protocol.

Resolution: Resolved. Buyers signatures were implemented.

5.3.3 Missing deadline for users' signatures on meta transactions

Severity: *Medium risk*

Context: NativeMetaTransactionFacet.sol#L44-L48

Description: The `NativeMetaTransactionFacet` diamond facet allows users to execute calls to the Kreatorhood NFT Marketplace using meta transactions. Users should sign an EIP712 object/message and then get their transaction processed by a relayer, passing signature verification:

```
IMetaTransaction.MetaTransaction memory metaTx = MetaTransaction({
    nonce: data.nonces[userAddress],
    from: userAddress,
    functionSignature: functionSignature
});

require(verify(userAddress, metaTx, sigR, sigS, sigV), "Signer and signature do not
    match");
```

A well-known problem when working with signatures is not including a `deadline` property to prevent users' signatures from being used far ahead in time, when users no longer want to execute the given action.

Recommendation: Consider implementing a signature deadline for meta transactions.

Resolution: Acknowledged. Currently, the only transactions executed via the `NativeMetaTransactionFacet` are cancel transactions where a deadline is not necessary. When the client offers other meta-transactions with more dangerous implications, they will consider adding a deadline for every meta-transactions.

5.4 Low risk

5.4.1 EIP712Domain struct has wrong order of arguments

Severity: *Low risk*

Context: ERC721MetaTx.sol#L10-L29, EIP712Base.sol#L7-L15

Description: The `EIP712Domain` struct has the following properties in the following order:

```
struct EIP712Domain {
    string name;
    string version;
    address verifyingContract;
    uint256 chainId;
}
```

However, the `EIP712_DOMAIN_TYPEHASH` has a different order of the `verifyingContract` and `chainId` arguments which may lead to integration issues:

```
bytes32 internal constant EIP712_DOMAIN_TYPEHASH =
    keccak256(bytes("EIP712Domain(string name,string version,uint256 chainId,address
        verifyingContract)"));
```

Recommendation: Switch the places of `verifyingContract` and `chainId` in the `EIP712Domain` struct declaration.

Resolution: Resolved.

5.4.2 Re-entrancy attack surface at multiple places throughout the codebase

Severity: *Low risk*

Context: ERC721B.sol#L210-L215, KreatorhoodERC721.sol#L50, FundsAdminFacet.sol#L40-L42, FundsAdminFacet.sol#L51-L55, MarketplaceFacet.sol#L54, MarketplaceFacet.sol#L127-L136, MarketplaceFacet.sol#L171-L175, MarketplaceFacet.sol#L248-L261

Description: There are numerous places throughout the NFT and Marketplace contracts where the Check-Effects-Interactions pattern is heavily violated, and callbacks are executed on untrusted addresses as buyers and sellers. However, no severe reentrancy attack vector was found during the review.

Recommendation: Consider adhering to the CEI pattern more strictly and avoiding any callbacks to untrusted users. Alternatively, implement reentrancy guards. If the latter is implemented, make sure to pay attention to the way reentrancy flags are stored and read because of the diamond pattern the codebase is following.

Resolution: Resolved. The client removed all “safe” transfers and mints.

5.4.3 Buyers will lose any excess native tokens sent when purchasing an NFT item

Severity: *Low risk*

Context: MarketplaceFacet.sol#L48, FundsAdminFacet.sol#L36-L60

Description: When buyers want to buy an NFT from the primary or secondary market, they can pay either with native tokens or ERC20 tokens (or both). However, the contracts do not check the actual msg.value that the buyer has sent but rely on the balance instead. This means that a user can send too many native tokens as msg.value by mistake and lose them because there’s neither a check nor a refund logic for excess native tokens.

Recommendation: Consider verifying that the msg.value is the same as the total amount of native tokens paid in the payment splits.

Resolution: Resolved.

5.4.4 Unnecessary payable modifier for function and constructor

Severity: *Low risk*

Context: ERC721MetaTx.sol#L69, KreatorhoodMarketplace.sol#L24

Description: The `ERC721MetaTx.executeMetaTransaction` function and `KreatorhoodMarketplace` diamond proxy’s constructor have the payable keyword/modifier, which is redundant as both methods do not deal with any native tokens.

Recommendation: Consider removing the payable modifiers.

Resolution: Resolved.

5.4.5 _msgSender() may return wrong value when a facet directly calls another one

Severity: *Low risk*

Context: ContextMixin.sol#L6

Description: The implementation of the `_msgSender` function looks as follows in the `ContextMixin` contract:

```
function _msgSender() internal view returns (address payable sender) {
    if (msg.sender == address(this)) {
        bytes memory array = msg.data;
        uint256 index = msg.data.length;
        assembly {
            // Load the 32 bytes word from memory with the address on the lower 20
                bytes, and mask those.
            sender := and(mload(add(array, index)), 0
                xffffffffffffffffffffffffffffffffffffffff)
        }
    } else {
        sender = payable(msg.sender);
    }
    return sender;
}
```

The contract assumes that if the call comes from address(this), then it must have been instantiated from the `NativeMetaTransactionFacet` facet, specifically the `executeMetaTransaction`:

```
(bool success, bytes memory returnData) = address(this).call(abi.encodePacked(
    functionSignature, userAddress));
```

However, the call can also be instantiated from another facet like the `MarketplaceBase` facet when calling functions directly on the `NativeMetaTransactionFacet`:

```
(bool valid, address signer) = IMetaTransaction(address(this)).
    tryRecoverSplitsSigner(
    splitsInfo,
    tradeDataHash
);
```

In this case, the msg.sender will not be appended to the calldata, and `_msgSender()` will read and return the wrong data from memory.

Recommendation: The issue is currently not severe because there are no instances of using `_msgSender()` after a call from one facet to another. However, make sure to be very careful when introducing new facets and functionalities and check for this issue.

Resolution: Acknowledged. Since this is only a potential issue that could escalate only when new code is added (no risk with the current version of the smart contracts), the client will review this when some future feature will require using the `_msgSender()` method.

5.4.6 NFT royalty fee should not be set at more than 10000 bps

Severity: *Low risk*

Context: MarketplaceFacet.sol#L114

Description: When minting a Kreatorhood NFT, a `royaltiesPercent` is passed, which is used for the EIP2981 royalty fee standard. However, a well-known issue is that it should not be possible for users to set these at 100% or more to avoid breaking external systems.

Recommendation: Consider adding a simple check verifying that the `royaltiesPercent` is not higher than 10000 bps.

Resolution: Resolved. The client implemented the recommended fix.