Sofamon V2 Security Review

Pashov Audit Group

Conducted by: ubermensch, Dan Ogurtsov, Alex Murphy

August 12th 2024 - August 15th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Sofamon/sofamon-v2-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Sofamon V2

Sofamon V2 is an innovative Gacha NFT Mint System that allows random minting through a commit-reveal scheme and introduces wearables as NFTs. It uses modular IRNG contracts like SupraRNG and ChainlinkRNG to generate randomness, ensuring unpredictable minting outcomes.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - [44f64fb0a0920cd11387d18e19e83fe6303dd89d]

fixes review commit hash - [355d37a537290225d81f6c11ff28a77e685f19a5]

Scope

The following smart contracts were in scope of the audit:

- `SofamonWearable`
- `SofamonExchange`
- `GelatoRNG`
- `SofamonWearableFactory`

7. Executive Summary

Over the course of the security review, ubermensch, Dan Ogurtsov, Alex Murphy engaged with Sofamon V2 to review Sofamon V2. In this period of time a total of **17** issues were uncovered.

Protocol Summary

Protocol Name	Sofamon V2
Repository	https://github.com/Sofamon/sofamon-v2-contracts
Date	August 12th 2024 - August 15th 2024
Protocol Type	NFT Mint System

Findings Count

Severity	Amount
High	3
Medium	3
Low	11
Total Findings	**17**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Signatures can be replayed using different addresses	High	Resolved
[H-02]	Use of msg.value inside a loop	High	Resolved
[H-03]	order.amount not accounted when filling	High	Resolved
[M-01]	Bypass blacklist using safeBatchTransferFrom and approval redirection	Medium	Resolved
[M-02]	Updating protocolFeeTo	Medium	Resolved
[M-03]	Sudoswap wrapper does not take into account refunds	Medium	Resolved
[L-01]	Changing the random generator contract might cause dos over honest commits	Low	Acknowledged
[L-02]	Incorrect ERC165 identifier for ERC2981	Low	Resolved
[L-03]	Solady's ECDSA implementation does not check for signature malleability	Low	Acknowledged
[L-04]	Use of transfer Instead of call	Low	Resolved
[L-05]	No way to revoke approval from blacklisted operators	Low	Resolved
[L-06]	No check for rarities.length	Low	Resolved
[L-07]	No sanity checks for rarities	Low	Resolved
[L-08]	Funds stuck if no response from	Low	Acknowledged

	Gelato		
[L-09]	Operators are not blacklisted	Low	Resolved
[L-10]	Griefing attack via onERC1155Received	Low	Resolved
[L-11]	Non-conformance to EIP-712 standard	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Signatures can be replayed using different addresses

Severity

Impact: Medium

Likelihood: High

Description

The `commitToMint` function in the contract allows signatures signed by the `signer` to be replayed using different addresses. Anyone can see the signatures on-chain and then replay them with different accounts that have the same nonce. An attacker can start inputting previous signatures submitted by a user starting from a nonce of 0 until the current one.

```
/// @param _collectionId The Id of the collection you wish to roll for
/// @param spins The amount of spins you wish to roll
/// @param signature A signature commit to the price, nonce, and minter from the
// authority address
///
/// @return ticketNonce The nonce to use to claim your mint
function commitToMint(
  uint256_collectionId,
  uint256spins,
  addressminter,
  bytesmemorysignature
) public payable returns (uint256 ticketNonce
    if (msg.sender != commitController) {
        uint256 nonce = userNonce[msg.sender];
        bytes32 hash = keccak256(abi.encodePacked
          (_collectionId, spins, nonce, msg.value, minter));
        bytes32 signedHash = hash.toEthSignedMessageHash();
        address approved = ECDSA.recover(signedHash, signature);

        if (approved != signer) {
            revert NotApproved();
        }
    }

    if (msg.value != 0) {
        payable(protocolFeeTo).transfer(msg.value);
    }

    ticketNonce = rng.rng();

    NonceData memory data = NonceData({ owner: minter, collectionId: uint128
      (_collectionId), spins: uint128(spins) });

    uint256 _userNonce = userNonce[msg.sender];
    userNonce[msg.sender] = _userNonce + 1;

    dataOf[ticketNonce] = data;

    emit MintCommited
      (msg.sender, minter, spins, msg.value, _userNonce + 1, ticketNonce);
}
```

Recommendations

To mitigate this issue, it is recommended to either include the address of the executor in the hash that is signed. This ensures that the signature is bound to a specific address and cannot be replayed by others. Here is a possible modification:

```
bytes32 hash = keccak256(abi.encodePacked
  (_collectionId, spins, nonce, msg.value, minter, msg.sender));
```

This addition ensures that the `msg.sender` address is included in the hash, preventing the signature from being valid for other addresses.

[H-02] Use of `msg.value` inside a loop

Severity

Impact: Medium

Likelihood: High

Description

The `batchCommitToMint` function attempts to call the `commitToMint` function multiple times within a loop, each time using the same `msg.value` that was provided when `batchCommitToMint` was called. While the `commitToMint` function works correctly on its own, it fails when invoked within the loop because the same `msg.value` is used multiple times. On the first iteration, the `msg.value` is transferred, but subsequent iterations fail because the `msg.value` has already been sent, causing the transaction to revert due to insufficient funds.

```
/// @dev Note, this function should only be called by the commitController,
// otherwise it will fail with no signature
/// @param mintData An array of all the spins to spin for
/// @return ticketNonces The nonces for each spin so that the user can claim
// their rewards
function batchCommitToMint
  (MintData[] calldata mintData) public payable returns (uint256[] memory ticketNonces
    ticketNonces = new uint256[](mintData.length);

    for (uint256 i; i < mintData.length;) {
        MintData memory data = mintData[i];

        uint256 nonce = commitToMint
          (data.collectionId, data.spins, data.minter, "");
        ticketNonces[i] = nonce;

        unchecked {
            ++i;
        }
    }
}

/// @param _collectionId The Id of the collection you wish to roll for
/// @param spins The amount of spins you wish to roll
/// @param signature A signature commit to the price, nonce, and minter from the
// authority address
///
/// @return ticketNonce The nonce to use to claim your mint
function commitToMint(
  uint256_collectionId,
  uint256spins,
  addressminter,
  bytesmemorysignature
) public payable returns (uint256 ticketNonce
    ...
    ...
    ...
    if (msg.value != 0) {
        payable(protocolFeeTo).transfer(msg.value);
    }
    ...
    ...
    ...
}
```

Recommendations

To mitigate this issue, consider adjusting both functions in a way that will not double account `msg.value` each iteration.

[H-03] `order.amount` not accounted when filling

Severity

Impact: High

Likelihood: Medium

Description

`fillAskOrder()` and `fillBidOrder()` operate with values equal to `order.price`. E.g.:

```
function fillAskOrder(FulfillOrder calldata order) public payable {
        uint256[] memory amounts = new uint256[](1);
        amounts[0] = order.amount;

        order.orderPool.swapTokenForSpecificNFTs{ value: order.price }
          (amounts, order.price, payable(msg.sender), false, address(0));

        emit AskFilled
          (order.orderPool, msg.sender, order.nftId, order.amount, order.price);
    }
```

In this case, order.amount should always be 1.

There is another function which executes multiple orders - `fulfillMultipleOrders()`, and the calculation is different there.

```
amounts, order.price * order.amount, payable
                      (msg.sender), false, address(0)
                );
```

This is the correct approach that allows trading with more than one token thanks to sending `order.price * order.amount`.

Recommendations

Consider accounting for `order.amount` in necessary fill functions.

8.2. Medium Findings

[M-01] Bypass blacklist using `safeBatchTransferFrom` and approval redirection

Severity

Impact: Medium

Likelihood: Medium

Description

The contract includes a blacklist feature intended to prevent certain addresses from transferring tokens. While the `safeTransferFrom` function checks both the sender and the recipient against the blacklist, the `safeBatchTransferFrom` function only checks the recipient. This allows blacklisted users to bypass the restriction and transfer their funds using `safeBatchTransferFrom`. Additionally, blacklisted users can redirect their approvals to another address since there is no blacklist check on the `msg.sender` in the approval functions.

Vulnerable Code

```
function safeTransferFrom(
  addressfrom,
  addressto,
  uint256id,
  uint256amount,
  bytescalldatadata
) public override NoBlacklist(to
    super.safeTransferFrom(from, to, id, amount, data);
}

function safeBatchTransferFrom(
    address from,
    address to,
    uint256[] calldata ids,
    uint256[] calldata amounts,
    bytes calldata data
)
    public
    override
    NoBlacklist(to)
{
    super.safeBatchTransferFrom(from, to, ids, amounts, data);
}
```

Recommendations

To address these issues, ensure that both the sender and the recipient are checked against the blacklist in the `safeBatchTransferFrom` function. Additionally, implement blacklist checks for the `msg.sender` functions to prevent blacklisted users from redirecting their approvals.

```
-    function safeTransferFrom
- (address from, address to, uint256 id, uint256 amount, bytes calldata data) public o
+    function safeTransferFrom
+ (address from, address to, uint256 id, uint256 amount, bytes calldata data) public o
        super.safeTransferFrom(from, to, id, amount, data);
    }

    function safeBatchTransferFrom(
        address from,
        address to,
        uint256[] calldata ids,
        uint256[] calldata amounts,
        bytes calldata data
    )
        public
        override
        NoBlacklist(to)
+       NoBlacklist(from)
+       NoBlacklist(msg.sender)
    {
        super.safeBatchTransferFrom(from, to, ids, amounts, data);
    }
```

[M-02] Updating protocolFeeTo

Severity

Impact: High

Likelihood: Low

Description

SofamonWearable has the following functions to update the fee receiver.

```
function updateRoyaltyFeeTo() public {
        royaltyFeeTo = ISofamonWearableFactory(machine).protocolFeeTo();

        emit NewRoyaltyFeeTo(royaltyFeeTo);
    }
```

So, `royaltyFeeTo` should always be updated manually.

Some bad scenarios are possible:

1. By default it is set to `address(0)` if the function was never called => royalties sent to zero address
2. `machine` can be updated => royalties sent to the deprecated address
3. `protocolFeeTo` can be updated in `machine` => royalties sent to the old `protocolFeeTo` address

Recommendations

Consider reading `ISofamonWearableFactory(machine).protocolFeeTo()` always, not storing the address on `SofamonWearable`. `updateRoyaltyFeeTo()` can be removed.

[M-03] Sudoswap wrapper does not take into account refunds

Severity

Impact: Medium

Likelihood: Medium

Description

The SofamonExchange contract includes wrapper functions like `fillAskOrder` and `fulfillMultipleOrders` to interact with Sudoswap. These functions facilitate purchasing NFTs by calling the `swapTokenForSpecificNFTs` function in the Sudoswap pool contract. However, the current implementation does not account for the possibility that `swapTokenForSpecificNFTs` might return a refund if more funds than necessary were sent. This omission will lead to loss of funds for the caller.

Reference: link

Recommendation

Modify the wrapper functions to account for potential refunds from the `swapTokenForSpecificNFTs` function.

8.3. Low Findings

[L-01] Changing the random generator contract might cause dos over honest commits

The `SofamonGachaMachine` contract includes a two-step process for minting wearable NFTs, involving a commit phase and a subsequent reveal phase. During the commit phase, a randomness request is made using an rng (random number generator) contract, and during the reveal phase, the random value associated with the previous request is retrieved from the rng contract.

The zSofamonGachaMachinez contract has a `setRNG` function that allows the contract owner to change the rng contract. However, if this function is called while there are pending commitments (i.e., commits that have not yet been revealed), those commitments will be unable to complete the reveal phase, effectively causing a Denial of Service (DoS) for honest users who have made those commitments.

Before allowing the rng contract to be changed, check if there are any pending commitments that have not yet been revealed. If there are pending commits, the `setRNG` function should revert the transaction and prevent the rng contract from being changed until all pending commits are revealed.

[L-02] Incorrect ERC165 identifier for ERC2981

The SofamonWearable contract implements the ERC165 standard, which is used to detect interface support in smart contracts. However, the contract incorrectly hardcodes the identifier for the ERC-2981 standard as `0x0e89341c`, whereas the correct identifier, as specified by the EIP-2981, is `0x2a55205a`.

Reference: link

Modify the contract code to replace the incorrect identifier `0x0e89341c` with the correct one `0x2a55205a` for ERC-2981 in the ERC165 implementation.

[L-03] Solady's ECDSA implementation does not check for signature malleability

Solady's ECDSA implementation does not include a check for signature malleability. Although this is not exploitable due to the nonce implementation in the current contract, it is advisable to handle this carefully to prevent users from producing another valid signature from the submitted ones.

Add a check to ensure that signatures are not malleable by enforcing a lower bound on the `s` value of the signature. This can be done by checking that `s` is less than or equal to `secp256k1n/2`, where `secp256k1n` is the curve order of the secp256k1 curve.

Example:

```
// Signature malleability check.
    if (uint256
      (s) > 0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff) {
        return false;
    }
```

Integrating this check into the contract will ensure that signatures cannot be manipulated to create another valid signature from the original one.

[L-04] Use of `transfer` Instead of `call`

The contract uses `transfer` to send Ether, which can lead to issues since `transfer` imposes a gas limit of 2300, which might not be sufficient if the recipient is a contract with complex logic in its fallback or receive functions. This can cause transactions to fail unexpectedly.

```
if (msg.value != 0) {
    payable(protocolFeeTo).transfer(msg.value);
}
```

Replace `transfer` with `call` to send Ether. This allows the recipient to receive more gas and avoids unexpected failures.

```
if (msg.value != 0) {
    (bool success, ) = payable(protocolFeeTo).call{value: msg.value}("");
    require(success, "Transfer failed");
}
```

By using `call`, you can ensure that the contract is more robust and can handle transfers to contracts with complex fallback or receive functions. Additionally, by checking the return value of `call`, you can detect and handle failed transfers more effectively.

[L-05] No way to revoke approval from blacklisted operators

`SofamonWearable.setApprovalForAll()` does not allow modifying approval for a given operator if it is blacklisted.

```
function setApprovalForAll
      (address operator, bool approved) public override NoBlacklist(operator) {
        super.setApprovalForAll(operator, approved);
    }
```

So, it is impossible to revoke approvals for blacklisted operators.

Consider removing `NoBlacklist(operator)` modifier for `setApprovalForAll()`.

[L-06] No check for rarities.length

`launchNewCollection()` sets rarities[] for the list of _wearables[]. But their lengths are not checked to be equal.

Later, `mintRandomNFT()` will iterate both rarities and wearables, using the same index:

```
function mintRandomNFT(
      Collectionmemorycollection,
      addressto,
      bytes32entropy,
      uint256ticketNonce
    ) internal {
      uint16 random = uint16(uint256(entropy)) % 10_000;

      for (uint256 i; i < collection.rarities.length;) {
          if (random < collection.rarities[i]) {
              uint256 wearableId = collection.wearables[i];
              sofamonWearables.mint(to, wearableId);

              emit NftMinted(ticketNonce, wearableId, to);
              return;
          }

          unchecked {
              ++i;
          }
      }
    }
```

In `launchNewCollection()`, consider checking that `_wearables.length == _rarities.length`

[L-07] No sanity checks for rarities

`launchNewCollection()` doesn't check `_rarities` values. In fact, there are two conditions expected:

1. Each `_rarities` value should be below or equal to 10_000
2. For the consistent probabilities, the value in `i` element should be higher than the value in `i-1` element.

Consider adding the checks above.

[L-08] Funds stuck if no response from Gelato

Randomness is always expected to be received from Gelato to mint wearables from the committed `msg.value` invested.

If Gelato is not responding, the funds for the given commit nonce will stuck waiting for the randomness provision until admins change VRF provider, or provide their own randomness.

Consider allowing users to withdraw committed `msg.value` if the randomness has not been provided for some meaningful time.

[L-09] Operators are not blacklisted

`SofamonWearable.setBlacklist()` can blacklist any address. `setApprovalForAll()` does not allow approval to blacklisted operators. It means that the blacklisted operators should not be allowed.

But if the approval was given before blacklisting, this operator will be able to send funds even being in the blacklist. The problem is that safeTransferFrom() and safeBatchTransferFrom() do not check that the `msg.sender` (operator) is blocklisted.

Consider adding modifiers `NoBlacklist(msg.sender)` for functions that transfer funds.

[L-10] Griefing attack via `onERC1155Received`

The `revealMintResult` function is used to mint an NFT based on the probability defined by the `mintOdds` of the collection. The minting function calls the `onERC1155Received` function of the receiver if it is a contract. However, a user can exploit this by reverting within the `onERC1155Received` function, preventing the minting process. This can be exploited to grief other users during the `revealMintResults` function, which mints NFTs for multiple users at once. A malicious user who is scheduled to receive an NFT can cause the entire transaction to fail, thereby preventing other users from receiving their NFTs.

```
//
// https://github.com/transmissions11/solmate/blob/97bdb2003b70382996a79a406813f76417b
function _mint(
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) internal virtual {
    ...
    require(
        to.code.length == 0
            ? to != address(0)
            : ERC1155TokenReceiver(to).onERC1155Received(msg.sender, address
              (0), id, amount, data) ==
                ERC1155TokenReceiver.onERC1155Received.selector,
        "UNSAFE_RECIPIENT"
    );
}
```

```
// Function in the contract
function revealMintResults(uint256[] calldata nonces) external {
    for (uint256 i; i < nonces.length;) {
        revealMintResult(nonces[i]);

        unchecked {
            ++i;
        }
    }
}
```

```
function revealMintResult(uint256 ticketNonce) public {
     ...
        if (_result < collection.mintOdds) {
            mintRandomNFT(collection, data.owner, keccak256(abi.encode
              (specificRandomness)), ticketNonce);
        } else if (_result < collection.mintOdds + collection.otherRewardOdds) {
            emit OtherResult(ticketNonce, data.owner, _result);
        } else {
            emit NoReward(ticketNonce, data.owner);
        }
     ...
}
```

To mitigate this issue, consider separating the minting process into individual transactions. This way, the failure of one user's minting will not affect the others. Another approach is to use `call` to invoke the `onERC1155Received` to prevent the failure of the transaction.

[L-11] Non-conformance to EIP-712 standard

The current implementation for hashing and signing messages does not conform to the EIP-712 standard for hashing and signing of typed structured data. This non-conformance results in several issues:

- Messages will not display correctly on wallets that support EIP-712.
- Signatures can be replayed across different contracts, chains, and functions due to the absence of chain ID, contract address, and function selector in the signed data.

```
if (msg.sender != commitController) {
    uint256 nonce = userNonce[msg.sender];
    bytes32 hash = keccak256(abi.encodePacked
      (_collectionId, spins, nonce, msg.value, minter));
    bytes32 signedHash = hash.toEthSignedMessageHash();
    address approved = ECDSA.recover(signedHash, signature);

    if (approved != signer) {
        revert NotApproved();
    }
}
```

Adjust the implementation to conform to the EIP-712 standard and mitigate the identified issues.