Sofamon Security Review

Pashov Audit Group

Conducted by: T1MOH, Dan Ogurtsov

February 26th 2024 - February 28rd 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-pwa-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

SofamonWearable is a Sofamon smart contract selling wearable collections with a bonding curve mechanism. Price grows according to the formula on every wearable purchased.

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 - 4f05651a7a7eb10a3a2d4fc34234fb55f08ac51b

fixes review commit hash - 01195301f92d054102c97e54c5c056f0db426fb1

Scope

The following smart contracts were in scope of the audit:

- `IBlast`
- `SofamonWearables`

7. Executive Summary

Over the course of the security review, T1MOH, Dan Ogurtsov engaged with Sofamon to review Sofamon. In this period of time a total of **11** issues were uncovered.

Protocol Summary

Protocol Name	Sofamon
Repository	https://github.com/Sofamon/sofamon-pwa-contracts
Date	February 26th 2024 - February 28rd 2024
Protocol Type	Bonding curve

Findings Count

Severity	**Amount**
High	1
Medium	4
Low	6
Total Findings	**11**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Rounding issue in price formula	High	Resolved
[M-01]	Using the same signature multiple times	Medium	Resolved
[M-02]	Excessive msg.value lost when buying	Medium	Resolved
[M-03]	Balance going below BASE_WEARABLE_UNIT	Medium	Resolved
[M-04]	Last user can't sell all his wearables	Medium	Resolved
[L-01]	abi.encodePacked is used in the hash calculation	Low	Resolved
[L-02]	Consider removing gas claim functions	Low	Resolved
[L-03]	Checks for fees	Low	Resolved
[L-04]	Max amount 2**(256/3), not 2**256	Low	Resolved
[L-05]	Checks for curveAdjustmentFactor	Low	Resolved
[L-06]	Creator to grief deals, on receiving fees	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] Rounding issue in price formula

Severity

Impact: Medium

Likelihood: High

Description

Here is current formula:

```
return (
            (totalSupply * curveFactor) / (totalSupply - x) -
              (totalSupply * curveFactor) / totalSupply
                - initialPriceFactor / 1000 * x
        ) * 1 ether;
```

There are several issues:

1. First parameter `(totalSupply * curveFactor) / (totalSupply - x)` is rounded down, because both numerator and denominator are 1e18 numbers.
2. Second parameter `(totalSupply * curveFactor) / totalSupply` can be simply reduced to `curveFactor`.
3. Third parameter `initialPriceFactor / 1000 * x` is always round down to 0 because `initialPriceFactor` is in range [100, 1000]. And also this parameter is meant to be in terms of 1e18, but previous parameters are just numbers. So in the end there will be underflow after fix.

Recommendations

Here are 1st and 2nd param precision increased to 1e18, and changed order of operations in 3rd param:

```
return (totalSupply * curveFactor * 1 ether) / (totalSupply - x) -
          (curveFactor * 1 ether) - initialPriceFactor * x / 1000;
```

8.2. Medium Findings

[M-01] Using the same signature multiple times

Severity

Impact: Medium, createSigner signs for the exact amount for a given sale, but in fact, it can be any amount in the end

Likelihood: Medium, easily available, requires receiving a signature only once for a private sale

Description

Private sales require a signature from `createSigner` for a given `wearablesSubject+amount`, with different signatures for buy and sell. But once signed the signature can be used many times. As a result, in fact, `createSigner` has no power to control amount for buy and sell operations. It is even possible to arrange a secondary market, where only one signature is used to access anyone to a sale, so the private market will be not so private.

Recommendations

Consider having a separate mapping for used signatures or applying an incrementing nonce logic in the signature.

[M-02] Excessive msg.value lost when buying

Severity

Impact: High, funds lost forever

Likelihood: Low, requires a mistake from a user

Description

`msg.value` is considered incorrect only in this equation:

```
msg.value < price + protocolFee + creatorFee
```

As a result, `msg.value` above price+fees will pass, but never return and thus lost.

It can happen in case of a mistake from users if they provide the wrong `msg.value`.

This mistake can be either from fee calculations, or it can be some sell operations before the user, so the total price is less than expected.

Recommendations

Consider either having a strict requirement to have msg.value the exact price+fees (better), or return excessive funds back to the user (more risky).

[M-03] Balance going below BASE_WEARABLE_UNIT

Severity

Impact: Medium, some balance cannot be sold and transferred, it is small but potentially valuable enough in case of low curveAdjustmentFactor or high supply

Likelihood: Medium, just required sending any number not strictly devisable by 0.001 ETH

Description

`transferWearables()` only requires sending amount above `BASE_WEARABLE_UNIT=0.001 ether`. But allows sending e.g. 0.0015 ether when the user has 0.002 ether. 0.0005 ether will remain in this case, below `BASE_WEARABLE_UNIT`.

The same thing for selling wearables.

```
function sellPrivateWearables
      (bytes32 wearablesSubject, uint256 amount, bytes calldata signature)
        external
        payable
    {
            // Check if amount is greater than base unit
            if (amount < BASE_WEARABLE_UNIT) revert InsufficientBaseUnit();
            ...
    }

    function transferWearables(
      bytes32wearablesSubject,
      addressfrom,
      addressto,
      uint256amount
    ) external {
        ...

        // Check if amount is greater than base unit
        if (amount < BASE_WEARABLE_UNIT) revert InsufficientBaseUnit();
        ...
    }
```

This new amount will not pass the minimum requirements in `transferWearables()` and sell operations. The only option for the user is to buy more wearables to pass the limit.

Recommendations

Consider checking that after transfer and a sell operation a user has the balance above `BASE_WEARABLE_UNIT`

[M-04] Last user can't sell all his wearables

Severity

Impact: Medium, user can be unable to sell if has only 1 share of Wearable

Likelihood: Medium, the last user to sell will get the error.

Description

Price calculation rounds down in both situations: buy and sell. Therefore buy price can be lower than the sell price if dividing one purchase into several batches, that's because of rounding down.

There is a possible situation when a lower price is paid on `buy` compared to the `sell`. As a result, the contract can calculate the sell price to be greater than

the buy price, as shown in this test:

```
function test_custom1() public {

// Setup wearable
// ------------------------------------------------------------
        bytes32 wearablesSubject = keccak256(abi.encode
          ("test hoodie", "hoodie image url"));

        vm.startPrank(signer1);
        bytes32 digest = keccak256(
            abi.encodePacked(
              creator1,
              "testhoodie",
              "hoodie",
              "thisisatesthoodie",
              "hoodieimageurl"
            )
        ).toEthSignedMessageHash();
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(signer1Privatekey, digest);
        bytes memory signature = abi.encodePacked(r, s, v);
        vm.stopPrank();

        vm.startPrank(creator1);
        sofa.createWearable(
            SofamonWearables.CreateWearableParams({
                name: "test hoodie",
                category: "hoodie",
                description: "this is a test hoodie",
                imageURI: "hoodie image url",
                isPublic: true,
                curveAdjustmentFactor: 50_000,
                signature: signature
            })
        );
        vm.stopPrank();
// ------------------------------------------------------------

        vm.startPrank(creator1);
        vm.deal(creator1, 1_000_000 ether);

        uint256 total = 0;
        // buy 10 batches of wearables
        for (uint256 i; i < 10; i++) {
            uint256 amount = 1e18 + 49_999;
            total += amount;
            uint256 buyPrice = sofa.getBuyPriceAfterFee
              (wearablesSubject, amount);
            sofa.buyWearables{value: buyPrice}(wearablesSubject, amount);
        }

        console.log("sellPrice                ", sofa.getSellPrice
          (wearablesSubject, total));
        console.log("SofamonWearables balance:", address(sofa).balance);

        // Sell all wearables
        //@note However it reverts with `error SendFundsFailed`, because not
        // enough balance in contract
        sofa.sellWearables(wearablesSubject, total);

        console.log(creator1.balance);
    }
```

The issue can be avoided if dividing sell on multiple batches, however, it is not possible if the user owns a minimal allowed amount of `BASE_WEARABLE_UNIT = 0.001 ether`

Recommendations

Round up when calculating the buy price.

8.3. Low Findings

[L-01] `abi.encodePacked` is used in the hash calculation

The same signature can be used to deploy multiple Wearables with different parameters because `abi.encodePacked` concatenates all the strings:

```
function createWearable(CreateWearableParams calldata params) external {
        // Validate signature
        {
            bytes32 hashVal = keccak256(
@>              abi.encodePacked(
  msg.sender,
  params.name,
  params.category,
  params.description,
  params.imageURI
)
            );
            bytes32 signedHash = hashVal.toEthSignedMessageHash();
            if (signedHash.recover(params.signature) != createSigner) {
                revert InvalidSignature();
            }
        }
      ...
    }
```

Here is a simple test:

```
function test_custom2() public {
        bytes memory string1 = abi.encodePacked("aabb", "");
        bytes memory string2 = abi.encodePacked("aa", "bb");
        assertEq(string1, string2);
    }
```

Recommendation:

```
-              abi.encodePacked
- (msg.sender, params.name, params.category, params.description, params.imageURI)
+              abi.encode
+ (msg.sender, params.name, params.category, params.description, params.imageURI)
```

[L-02] Consider removing gas claim functions

In the constructor you set Governor. According to Blast documentation, when Governor of address is set, only Governor is allowed to claim gas and yield. Therefore these functions are redundant: `claimAllGas()`, `claimMaxGas()`, `claimGasAtMinClaimRate()`

In case you consider returning Governance to contract `SofamonWearables`, you should implement access control and claim of yield

[L-03] Checks for fees

SofamonWearables.sol does not check fee setup, so any numbers are possible. But there are some bad values, e.g. sum of fees above 100% - in this case, it will be possible to buy Wearables, but sell will revert.

Consider additional checks when setting fees:

- setting a limit for `protocolFeePercent`
- setting a limit for `creatorFeePercent`
- the sum of two fees below 100% (or some smaller X)

[L-04] Max amount `2**(256/3)`, not `2**256`

Curve calculation makes a secure arithmetics - division after multiplying. And the first multiplying is `amount**3`.

It sets an overflow limitation of `2**(256/3)` for amount. On the other hand, README describes `amount` as - `is the total supply of the wearables, which can range from 0 to the maximum limit of uint256`

So either the README should indicate this limit or the price math should be adjusted to allow larger numbers.

[L-05] Checks for curveAdjustmentFactor

Setting `curveAdjustmentFactor` does not have any checks. In case of very high numbers, it can cause free wearable purchases without fees.

Consider setting min and max limits for `curveAdjustmentFactor`.

[L-06] Creator to grief deals, on receiving fees

`SofamonWearables` distributes ETH fees using raw calls. This is the opportunity for all receivers to affect the behavior of buy and sell operations.

A malicious actor can be `wearables[wearablesSubject].creator` - the creator of a given `wearablesSubject`. The creator can track if a given fee is received from a buy or sell operation (checking for an increase or decrease of `wearablesSupply[]`), and e.g. revert on sell operations.

It is better to have a separate mapping to store pending ETH fees so that creators will have to call a separate function to withdraw fees.