PassOnCore

Security Review

6th September, 2023

Review Conducted by:
Lyuboslav Bardenski, Smart Contract Security Researcher

Disclaimer

This smart contract security review does not bring any additional responsibilities for the group of people who have conducted it. The goal of the smart contract security review is to find possible risks and vulnerabilities, and provide useful recommendations in terms of security and optimization.

Severity classification

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

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 behavior that's not so critical.

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.

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.

Security Assessment Summary

Name	PassOnCore
Repository	https://github.com/passon-io/contracts
Review Commit Hash	4c2854cf69f1ac6c6f4e8c9132fd8c62624cf3ae
Resolution Commit Hash	ed11b7ba1765916e4e0f1a2d7f2c12033d342fe9

Scope

The following smart contracts were in scope of the audit:

- *core/contracts/PassOnCore.sol*

Critical	1
High	1
Medium	4
Low	2
Informational	5

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the PassOn smart contracts.

Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labeled as “informational”.

Each vulnerability is also assigned a status:

• Open: the issue has not been addressed by the project team.

• Resolved: the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.

• Partially Resolved: the issue was acknowledged by the project team but not the entire vulnerability was fixed.

Findings Summary

ID	Description	Severity
C-01	Signature malleability in `manageFunds()`	CRITICAL
H-01	`publicKey` can increase _referrerFunds and _refereeFunds	HIGH
M-01	Functions are not compatible with tokens with missing return value	MEDIUM
M-02	Functions are not compatible with fee on transfer tokens	MEDIUM
M-03	Referee can register with infinite count of referral codes causing out of gas in `withdrawAllFundsForReferee`	MEDIUM
M-04	Referrers can register with infinite amount of referral codes causing out of gas in `withdrawAllFundsForReferrer`	MEDIUM
L-01	`publicKey` cannot be changed if it is compromised	LOW
L-02	`campaignId` is not checked for 0 value	LOW
I-01	Store data from storage *in memory* in order to save gas from ***sload*** operations	INFORMATIONAL
I-02	Redundant for loop	INFORMATIONAL
I-03	`manageFunds` can be called with infinite amount of values causing out of gas in `manageFunds`	INFORMATIONAL
I-04	Pack structs to optimize gas	INFORMATIONAL
I-05	Require statements can be reused in modifiers or functions to decrease contract size	INFORMATIONAL

C-01 Signature malleability in *manageFunds()*

SEVERITY | CRITICAL
LIKELIHOOD | HIGH
IMPACT | HIGH

Description

`*manageFunds()*` function takes `_users`, `_entityTypes`, `_referralCodes`, `_amounts`, `_numRewardableActions`, `unsignedMessage`,`_r`, `_s`, `_v` as arguments but none of them is formed in a standardized structure and hashed, as it is by the ***EIP712*** standard. `*verify()*` function is never used in the contract.

The signature does not contain also the following indispensable arguments: ***nonce, deadline, chainId, verifyingContract***.

line 343

```
        require(
            recoverSigner(
                keccak256(abi.encodePacked(unsignedMessage)),
                _r,
                _s,
                _v
            ) == publicKey,
            "Invalid signature. Could not verify source as Pass On Funds Manager."
        );
```

line 673

```
    function recoverSigner(
        bytes32 _hash,
        bytes32 _r,
        bytes32 _s,
        uint8 _v
    ) internal pure returns (address) {
        bytes32 messageHash = keccak256(
            abi.encodePacked("\x19Ethereum Signed Message:\n32",
_hash)  );
        return ecrecover(messageHash, _v, _r, _s);
    }
```

Due to the fact that the signature does not validate any of the aforementioned arguments nor having it poses a critical impact on the contract's security. Anyone can execute `manageFunds()` with the same signature with any parameters which would lead to draining the contract's funds.

Recommendations

There are two possible solutions for this vulnerability:

1. Follow ***EIP712*** standard and include all of the function arguments in typed data structure. All the domain hash parameters must be included, as also a nonce must be added in order to guarantee that signature won't be used again.

   In order to recover the signature use the correct function from te OpenZeppelin's ***ECDSA*** library.

2. Remove signature verification and create access control modifier in the `manageFunds` function to be only callable by the `publicKey`.

Resolution | Resolved

The client has implemented the second approach.

H-01 publicKey can increase referrer and referee funds infinitely

SEVERITY | HIGH
Likelihood | MEDIUM
Impact | HIGH

Description

In the **`manageFunds()`** function if the publicKey can sign a message with unlimited token amounts to a specific address, which will effectively lead to improperly distributed funds which could not be repaired.

Recommendations

Create validation checks for a maximum amount with which referrer and referee balances can be increased based on their rewards.

Resolution | Resolved

Rules for allocation of the funds between referee and referrer have been set in order to ensure that the distributed funds are in a proper boundaries.

M-01 Functions are not compatible with tokens with missing return value

SEVERITY | MEDIUM
Likelihood | LOW
Impact | HIGH

Description

Functions ***registerCampaign(), depositFunds(), withdrawFundsForCampaign(), withdrawAllFundsForReferrer()*** and ***withdrawAllFundsForReferee()*** call `ERC20.transfer()` and `ERC20.transferFrom()` and expect a bool value.

Some tokens do not return a bool (e.g. USDT, BNB, OMG) on ERC20 methods. Since the require-statement expects a bool value, for such a token a void return will also cause a revert, despite an otherwise successful transfer. That is, the token payout will always revert for such tokens.

Recommendations

Use OpenZeppelin's ***SafeERC20***, which handles the return value check as well as non-standard-compliant tokens or limit the supported tokens in the contract.

Resolution | Resolved

OpenZeppelin ***SafeERC20*** was integrated by PassOn.

M-02 Functions are not compatible with fee on transfer tokens

SEVERITY | MEDIUM
Likelihood | LOW
Impact | HIGH

Description

Functions registerCampaign(), depositFunds() call `ERC20.transfer()` and `ERC20.transferFrom()` without checking balance before and balance after which leads to incorrect accounting.

Recommendations

Check the balance before and after the transfer to take into account the Fees-On-Transfer or limit the supported tokens in the contract.

Resolution | Resolved

M-03 Referee can register with infinite amount of referral codes causing out of gas in `withdrawAllFundsForReferee`

SEVERITY | MEDIUM
LIKELIHOOD | LOW
IMPACT | HIGH

Description

Referees can register an infinite amount of referral codes in `*registerWithReferralCode()*` causing *`withdrawAllFundsForReferee()`*.

Recommendations

Create validation for a maximum number of registrations per user.

Resolution | Resolved

M-04 Referrers can register with infinite amount of referral codes causing out of gas in `withdrawAllFundsForReferrer`

SEVERITY | MEDIUM
Likelihood | LOW
Impact | HIGH

Description

Referrers can generate an infinite amount of referral codes in `*generateReferralCode()*` causing *`withdrawAllFundsForReferrer()`*.

Recommendations

Create validation for a maximum number of generated codes per user.

Resolution | Resolved

L-01 `publicKey` cannot be changed if it is compromised

SEVERITY | LOW
Likelihood | LOW
Impact | LOW

Description

`publicKey` is only set in the constructor function of the PassOnCore contract. If the private key of the account is compromised and the `publicKey` can't be changed by the owner, it means that the contract won't be functional again as the only one option left is to pause the contract.

Recommendations

Create a setter function for `publicKey` with `onlyOwner` modifier.

Resolution | Resolved

L-02 `campaignId` is not checked for 0 value

SEVERITY | LOW
Likelihood | LOW
Impact | LOW

Description

Campaign ids start from 1 as it is explicitly set in the contract, but the functions `generateReferralCode()`, `depositFunds()` and `toggleCampaignState()` do not perform validation if the campaign id is zero as the only check is made in `withdrawFundsForCampaign()`.

Recommendations

Add validation in each of the `generateReferralCode()`, `depositFunds()` and `toggleCampaignState()` like it is performed in `withdrawFundsForCampaign()`.

```
require(
    _campaignId > 0 && _campaignId < nextCampaignId,
    "Invalid campaign id"
);
```

Resolution | Resolved

I-01 Store data from storage in memory in order to save gas from sload operations

Description

In each iteration sload opcode is used to get the length of the `_msgSender()` registrations.

line 293

```
for (uint256 i = 0; i < referees[_msgSender()].length; i++) {
```

Recommendations

Store in memory variable `referees[_msgSender()].length`.

Resolution | Resolved

I-02 Redundant for loop

Description

The for loops on line 191 and line 231 can be packed into one in order to save gas from redundant iterations.

Recommendations

```
for (uint256 i = 0; i < _rewards.length; i++) {
      BrandCampaignRule memory ruleDef = BrandCampaignRule({
         reward: _rewards[i],
         contractAddress: _contractAddresses[i],
         chainId: _chainIds[i],
         eventName: _eventNames[i],
         partialAbi: _partialAbis[i],
         criteriaType: _criteriaType[i],
         criteria: _criteria[i]
       });

       newCampaign.ruleDefinitions.push(ruleDef);

       emit BrandCampaignRuleRegistered(
            i,
            nextCampaignId,
            _rewards[i],
            _contractAddresses[i],
            _chainIds[i],
            _eventNames[i],
            _partialAbis[i],
            _criteriaType[i],
            _criteria[i]
        );
}
```

Resolution | Resolved

I-03 `manageFunds` can be called with infinite amount of values causing out of gas in `manageFunds`

Description

`manageFunds` does not perform any checks for the length of the input arrays which can cause out of gas.

```
    function manageFunds(
        address[] memory _users,
        EntityType[] memory _entityTypes,
        string[] memory _referralCodes,
        uint256[] memory _amounts,
        uint256[] memory _numRewardableActions,
        string memory unsignedMessage,
        bytes32 _r,
        bytes32 _s,
        uint8 _v
    ) external nonReentrant whenNotPaused {
        require(
            _users.length == _entityTypes.length &&
                _entityTypes.length == _referralCodes.length &&
                _referralCodes.length == _amounts.length &&
                _users.length > 0,
            "Array(s) empty or lengths do not match (input params)."
        );

        require(
            recoverSigner(
                keccak256(abi.encodePacked(unsignedMessage)),
                _r,
                _s,
                _v
            ) == publicKey,
            "Invalid signature. Could not verify source as Pass On Funds
Manager.");

        for (uint256 i = 0; i < _users.length; i++) {
            ...
        }
    }
```

Recommendations

Add validation for the maximum length of the input arrays.

Resolution | Resolved

I-04 Pack structs to optimize gas

Description

It is possible to store as many variables into one storage slot, as long as the combined storage requirement is equal to or less than the size of one storage slot, which is 32 bytes. For example, one bool variable takes up one byte. A uint8 is one byte as well, uint16 is two bytes, uint32 four bytes, and so on.

Recommendations

Pack in a proper gas optimized way the structs `Funds`, `BrandCampaignRule` and `BrandCampaign`.

Recommendation example for `*Funds*`:

```
struct Funds {
    uint256 amount;
    address tokenAddress;
}
```

Resolution | Resolved

I-05 Require statements can be reused in modifiers or functions to decrease contract size

Recommendations

Reuse the checks for valid `*campaignId*`, campaign registrant and if a campaign is active.

Resolution | Partially Resolved