CoinFabrik

sorare

Short Code Review

04-May-2020

By CoinFabrik

Introduction

CoinFabrik was asked to conduct a **short code review** of the smart contracts written for the **Sorare** project. Due to the haste, many deep analyses that are usually part of a formal audit process had been ruled out. On the other hand, some of the findings we describe below could be false positives since they have not been tested in a simulated environment. They are included in this report and should be understood as red flags to be confirmed. First we will provide a summary of our findings and then we will show more details.

Summary

The contracts reviewed are from the **Sorare** repository at GitLab: https://gitlab.com/sorare/blockchain/-/tree/master/src/contracts.

This review is based on the commit `a76b43282e630dcc56198fd37ca453756c4b818d`, and updated to reflect changes at `bac72c11e985893c64af9e677eab54d225ebdbcf`.

Our Findings

1. Sender's Replay Attack (Critical)
2. Receiver/Sender ownership verification logic issue (Critical)
3. Invalid signature verification for token transfer (Critical)
4. Relay contract replay attack(Critical)
5. Jeopardized ETH amount validation by uint overflow (Medium)

Scope of this Code Review

The the following contracts were reviewed:

1. Bank.sol
2. BatchMinter.sol
3. CapperAccess.sol
4. CashDesk.sol
5. Closable.sol
6. IBank.sol
7. INextContract.sol
8. ISignerAccess.sol
9. ISorareCards.sol
10. ISorareData.sol

11. ISorareTokens.sol
12. Migrations.sol
13. MinterAccess.sol
14. NFTClient.sol
15. Relay.sol
16. RelayRecipient.sol
17. Signature.sol
18. SignerAccess.sol
19. SorareCards.sol
20. SorareData.sol
21. SorareMetaProxy.sol
22. SorareTokens.sol
23. Taxable.sol
24. mocks/ERC721Holder.sol
25. mocks/ERC721MintableBurnableImpl.sol
26. mocks/ExposedTaxable.sol
27. mocks/NextContract.sol
28. mocks/SignerAccessWithBankApproved.sol

Analyses

The following analyses were performed:

- Misuse of the different call methods: call.value(), send() and transfer().
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.
- Old compiler version pragmas.
- Race conditions such as reentrancy attacks or front running.
- Misuse of block timestamps, assuming anything other than them being strictly increasing.
- Contract softlocking attacks (DoS).
- Potential gas cost of functions being over the gas limit.
- Missing function qualifiers and their misuse.

- Fallback functions with a higher gas cost than the one that a transfer or send call allows.
- Fraudulent or erroneous code.
- Code and contract interaction complexity.
- Wrong or missing error handling.
- Overuse of transfers in a single transaction instead of using withdrawal patterns.
- Insufficient analysis of the function input requirements.

Detailed findings

Severity Classification

The security risk findings are evaluated according to the following classification:

- **Critical:** These are issues that compromise the system seriously. We suggest fixing them **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to exploit but can be used in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.
- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

Severity	Exploitable	Production roadblock	We suggest fixing it
Critical	Yes	Yes	Immediately
Medium	Potentially	Yes	As soon as possible
Minor	Unlikely	No	When possible
Enhancement	No	No	Optionally

Special Severity Classification Criteria

All findings are potential and no certain or tested results are available.

Fund losses are mitigated because ultimately the withdrawal of ETH is handled in the cashdesk, under centralized approval, and token fraud can be returned using **RelayAddreess** account on the stealer behalf.

However, the severity criteria was determined according to the trust in the logic of deal validation and user's authentication signature verification.

Critical severity

Sender's Replay Attack

Assumption: we understand that a sender can sign an ***"open receiver deal"*** and a receiver can take that deal, sign it, and execute it.

If our assumption in the signing process is correct, then a malicious sender can listen to the transaccion mempool for that **DEALID** settle execution, to move just enough funds or assets, the moment before, paying higher gas price, to make the deal unable to be settled.

The whole purpose is to capture the signed deal from the failed transaction sent by the receiver and, having the captured message, the attacker could replay the settlement at a future moment for example when the token is devalued, due to decreased scarcity, getting more eth for what is the token now worth. Forcing the receiver to pay the price that signed for a, now devalued, token.

This problem relies on the inability of the receiver to cancel a deal unlike the sender, who can cancel a deal at any time using the function **cancelDeal**:

```
    function cancelDeal(uint256 _dealId) external {
        deals[msg.sender][_dealId] = true;
        emit BankDealCancelled(_dealId, msg.sender);
    }
```

Given that the sender and the receiver are both essentially the same users (since they can both send and receive tokens and ETHs), both should be able to cancel a signed deal.

Solution:

Add an additional require in the **settle** function such as:

```
require(
    !deals[_deal.receiver][_deal.dealId],
    "Deal settled or cancelled"
);
```

Now the receiver can use the function mentioned above, the same as the sender.

Receiver/Sender ownership verification logic issue

This attack relies on the logic that verifies that the receiver/sender owns the token they are trying to trade.

Several attacks can be derived from this flaw but we will describe only one, which consist in offering certain token in the deal and accomplishing a successful **settle** when the token was not actually sent.

Here is the logic that checks the receiver owns the token (similar logic applies to sender).

```
internalTokensIndex = 0;
mappedTokensIndex = 0;
for (uint256 i = 0; i < _deal.receiveTokenIds.length; i++) {
    uint256 token = _deal.receiveTokenIds[i];
    address owner = nonFungibleContract.ownerOf(token);
    if (owner == receiveInternalTokens.from) {
        receiveInternalTokens.tokenIds[internalTokensIndex++] = token;
    } else {
        receiveMappedTokens.tokenIds[mappedTokensIndex++] = token;
    }
}
require(
    internalTokensIndex + mappedTokensIndex ==
        _deal.receiveTokenIds.length,
    "Receiver doesn't own all tokens"
);
```

Here the **_deal.receiveTokenIds.length** will always be **internalTokensIndex + mappedTokensIndex** as they are incremented either one or the other on each iteration for being in an **'if/else'** clause.

Therefore, it makes the verification invalid for every case. This is not such a big problem because the actual token ownership verification is made ultimately in the **transferFrom** function of the token itself and, if it is not the actual owner, the whole transaction would be reverted.

Solution:

Have two separate **'if'** clauses, one for each verification, instead of the current **'else/if'** clause. A possible code could look like this:

```
 if (owner == receiveInternalTokens.from) {
     receiveInternalTokens.tokenIds[internalTokensIndex++] = token;
 }
 if (owner == receiveMappedTokens.from){
     receiveMappedTokens.tokenIds[mappedTokensIndex++] = token;
 }
```

Invalid signature verification for token transfer

Note: the code here reflects the changes made to Fix settling a Deal with no tokens.

Here the vulnerability is in the condition that **&& (tt.tokenIds[0] != 0)**

If that condition is **false** then none of the tokens are transfered.

```
for (uint256 tti = 0; tti < _tokenTransfers.length; tti++) {
    TokenTransfer memory tt = _tokenTransfers[tti];

    if (tt.tokenIds.length > 0 && tt.tokenIds[0] != 0) {
        Signature.requireSignature(
            abi.encode(tt.sigType, dealBlob),
            tt.signature,
            tt.from
        );

        address tokenRecipient = tt.to;
        if (useMappedAccountAsDefault[tt.to]) {
            tokenRecipient = mappings[tt.to];
        }

        nonFungibleContract.transferTokens(
            tt.from,
            tokenRecipient,
            tt.tokenIds
        );
    }
}
```

A sender could offer a deal where the token tokenId=0 is the first token, followed by his other tokens. Then a bidder would sign the deal expecting to receive the offered tokens and naively ignoring the first **tokenIds[0] = 0** and settle it. But none of the tokens offered by the sender would be transferred to the receiver. The sender would keep all his tokens and also get receiver's funds and tokens, for free.

Solution:

Remove the condition **(tt.tokenIds[0] != 0)** from the **'*if*'** clause, leaving just:

```
if (tt.tokenIds.length > 0 ) {
```

Relay contract replay attack

Relay.sol

By using a middleware contract we can call the function **purgeBatchNonces** multiple times in the same transaction resulting in the eventual reset of the variable **_batchNonces[sender]** allowing us to reuse a signed transaction for a previous nonce. How many times this function needs to be called by the attacker's contract depends on the value of **_batchSize** and subsequently, the gas cost to execute the attack which, for a relatively small **_batchSize**, the attack could be too expensive. However, it should be considered that **delete _burnedBatchNonces[sender]** may return some of the gas cost to the attacker contract caller. And that the transaction to replay may be, for example, a very valuable token transfer making the attack much more cost effective. BUT even so, the transaction gaslimit may result in a ridiculous amount of transactions needed to accomplish the attack and the time it would require being too high.

```
    function purgeBatchNonces(address sender) public onlySigner {
        delete _burnedBatchNonces[sender];
        _batchNonces[sender] = _batchNonces[sender] + _batchSize;
    }
```

A variant of this attack consists of a denial of service of the **RelayBatchCall** function by calling the **purgeBatchNonces** in the middle of its execution preventing it from finishing.

Solution:

This function should require a signature from the sender allowed to purge the batch nonces. A possible resulting code could have the following shape:

```
    function purgeBatchNonces(
        address sender,
        uint256 batchNonce,
        bytes memory signature) public onlySigner {

        require(batchNonce=_batchNonces[sender],"Invalid Nonce")
        bytes memory message = abi.encode(
            batchNonce,
            address(this)
        );
        Signature.requireSignature(message, signature, sender);
        delete _burnedBatchNonces[sender];
        _batchNonces[sender] = _batchNonces[sender] + _batchSize;
    }
```

Medium severity

Jeopardized ETH amount validation by uint overflow

```
function depositETH(
    address[] calldata _addresses,
    uint256[] calldata _amountsInWei
) external payable {
    require(
        (_addresses.length == _amountsInWei.length),
        "Incorrect arguments"
    );
    uint256 requiredAmount = 0;
    for (uint256 i; i < _amountsInWei.length; i++) {
        requiredAmount += _amountsInWei[i];
    }
    require(msg.value == requiredAmount, "Value sent does not match");

    for (uint256 i; i < _addresses.length; i++) {
        sorareBankContract.depositETH.value(_amountsInWei[i])(
            _addresses[i],
            "",
            false
        );
    }
}
```

The **depositETH** function uses a **'for'** statement to add all **amountInWei** from the integer array used as parameter in the **requiredAmount** variable and then compares this value with **msg.value** using a **'require'** as validation. The problem is that the operation is not done using **SafeMath** library allowing sending different possible combination of integer values in the **amountInWei** array that adding them to the **requiredAmount** variable one after the other in each iteration will make the variable overflow consequently causing the validation (in the **'require'**) to fail since it will be equal to **msg.value** but it will not represent the real total of the sum of **amountInWei**.

To accomplish this attack, a specific combination of values, in both arrays, should be used as parameters. Even though the impact of this vulnerability was not analyzed, the fact that the total ETH validation can get jeopardized and then used to interact with the bank, exposes a big risk that might be combined with other bugs or escalated to a more complex attack.

Solution:

Apply **SafeMath** library instead of the **+=** operation.

```
requiredAmount = requiredAmount.add(_amountsInWei[i]);
```

Minor severity

Not found yet.

Enhancements

Not found yet.

Conclusion

Given the short time we had to examine the code, it is not defined yet but in general terms, we can say it is a great project with big potential and very good code quality.

Once fixed the identified issues, we think it is safe to launch. However we suggest requesting a full audit of the code looking for any door that might remain open after the changes, and any bug that might have remained unseen in this review.