SOLIDIFIED

Audit Report for FansIUnite January 17, 2019.

Summary

Audit Report prepared by Solidified for FansUnite covering the the fansunite-core smart contract repository.

Process and Delivery

Two independent Solidified experts performed an unbiased and isolated audit of the below smart contracts. The debriefing took place on January 17, 2019.

Audited Files

The following files were covered during the audit:

- BetManager.sol
- LeagueRegistry.sol
- Registry.sol
- ResolverRegistry.sol
- Vault.sol
- FanToken.sol
- ERC165.sol
- ChainSpecifiable.sol
- RegistryAccessible.sol
- BaseLeague.sol
- League001.sol
- LeagueLib001.sol
- LeagueFactory001.sol
- BetLib.sol
- ExposureLib.sol
- SignatureLib.sol
- RMoneyLine2.sol
- RSpreads2.sol
- RTotals2.sol
- BaseResolver.sol

Notes

The audit was based on the commit hash `6f177385e41b505e4fac02ce6a69674bee8725ae`, solidity compiler `0.4.24+commit.e67f0147`

Issues Found

Moderate

1. Upgrading the vault contract in Registry.sol will cause previous stakes by both backers and layers inaccessible by the contract

The Vault contract is exchangeable by design in `Registry.sol`, and its address is queried (also on registry.sol) every time a transaction is performed. If the vault is replaced while there are active leagues, all registered bets in these leagues will be rendered unclaimable.

The current version is safe, as long as the vault contract is only replaced when it holds no value.

Recommendation
We recommend making sure the vault is only replaceable when there are no active leagues (and no funds held by the vault).

Minor

2. Moneyline resolver does not implement segmentation as specified in IResolver.sol

The Moneyline resolver does not implement segmentation as specified in `IResolver.sol`:

10. The segmenting function MUST consume all the same parameters as the validator function
11. The segmenting function MUST return a bytes32, for each segment, per fixture
13. Segment must be unique within a single fixture for the given resolver.

The moneyline resolver is currently returning a fixed value, fact that will impact exposure calculation (This bug was brought to our attention by fansUnite). The contract's specification about segmentation, is also incorrect, reflecting current implementation.

Additionally, the `segment` function does not take all the arguments the validate function takes.

Recommendation
Review `segment` function from `MoneyLine2.sol` (and it's sepcification) to ensure it is compliant with IResolver specifications. Consider removing the comment "// NOTE This is a HIGHLY experimental resolver, do NOT use in production" from `RSpreads2.sol` and `RTotals2.sol`.

3. The Spender is approved to manipulate the user's whole balance in Vault.sol

When a user approves spending by the `betManager`, the whole balance becomes manipulable by it, irrespective of the bet values or user balance.

Recommendation
It is strongly recommended that the approve function is refactored to include a cap. This will not only provide additional security to users but also reduce the impact of an attack in case an approved spender is compromised.

4. Loop included in _areParticipants could reach the block gas limit, causing Denial of Service on scheduleFixture

The for loop in `_areParticipants` is bounded by the (`participantsPerFixture` of the league) * (total participants of the league). While this is not a problem for leagues that have a lower number of participants (i.e. soccer), it could reach the block gas limit for large leagues with a high number of participants per fixture (marathon or surf are possible examples). This can cause scheduleFixture to be unusable, effectively preventing that league to be used within the platform

Recommendation
We recommend refactoring the `_areParticipants` function to account for block gas limits, or bounding the number of participants and participants per fixture of a league to ensure `_areParticipants` executes within current limits.

5. Address keys are always added as valid in Registry

A mapping of valid address keys is maintained in the `Registry` contract:
`mapping (bytes32 => bool) private validAddressKeys;`
However, on address registration, the key is always added to this mapping (and never removed).

Recommendation
As all keys are valid, this extra mapping is unnecessary and the following code does not add any functionality:
`require(validAddressKeys[keccak256(bytes(_nameKey))], "Namekey must be valid");`

6. RegistryAccesible contract does not use Ownable features

The `RegistryAccesible` contract inherits from `Ownable` but does not use any ownership feature. All other contracts that inherit from `RegistryAccesible` also inherit from `Ownable`, making the repeated inheritance unnecessary.

Recommendation
The is Ownable declaration should be removed either in the base contract or in the derived contracts.

7. Domain separator is not editable, enabling cross-chain attacks in case of hard fork

All contracts inherit from ChainSpecifiable.sol, allowing the chain Id to be changed in case of a hard fork of the Ethereum chain, in order to prevent cross-chain replay attacks. The contract betManager.sol, however, also make use of a domain separator (EIP712) for that purpose.

The domain separator is created during deployment, in the BetManager.sol constructor, and is not editable later on. The domain separator is used to verify bets are targeting the correct chain, but it's not editable in case of a hard fork. If chainId in chainSpecifiable is edited it has no impact in the Domain Separator.

Recommendation
We recommend that the domain separator is generated dynamically, every time the _chainId is updated, to ensure consistency between the two and effectively prevent cross-chain attacks.

Notes

8. Use Open Zeppelin for signature validation

The contracts implement their own signature validation functionality. However, Open Zeppelin,. which is used for other functionality already includes signature validation code: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/cryptography/ECDSA.sol

Recommendation
We recommend using Open Zeppelins version, as it is widely used (and tested) by the community.

9. The contracts accept any user-provided token address

Contracts currently accept bets for any given token (as long as both the backer signs the bet and the layer submits it). A rogue contract can be used by both backer and layer, effectively allowing them to play with no financial staking (except for gas costs). Furthermore, the user provided addresses are called a number of times by the vault.sol contract, effectively transferring the control flow to it.

Recommendation
It is recommended to restrict the tokens allowed to a select list of widely used and verified ERC20 contracts.

10. Renounce Ownership can cause DoS if called on any of the contracts

Several contracts use Open Zeppelin's `Ownable` as a way to control access to privileged functions within the contracts. These privileged functions range from changing a smart contract (effectively upgrading it) in one of the registry contracts to upgrading classes, leagues and participants in Leagues.

While there are use cases for renouncing ownership of a contract, the in-scope contracts do not specify such a situation, resulting in an unnecessary increase of the attack surface.

Recommendation
Consider overloading the `renounceOwnership` and disabling it, or at least requiring a transfer (waiting to be claimed, per the latest Ownable contract version) before the renouncement is executed.

11. Use of deprecated keyword "var"

The keyword var has been deprecated since Solidity 0.2.20. Its use has a number of issues, the main one being that it will choose the type on execution time depending on the value it was provided, the fact that can cause unexpected reverts depending on how this variable is used later on. It is currently used in all resolver contracts, as follows:

```
var (, _participants,) = ILeague001(_league).getFixture(_fixture);
```

Recommendation
We recommend updating the contracts and declaring the used variable in it's expected type, as follows:

```
(,uint256[] memory _participants,) =
ILeague001(_league).getFixture(_fixture);
```

12. Compiler and libraries outdated, experimental features

Solidity compiler 0.4.24 and Openzepellin-solidity 2.0 rc1 were used in the in-scope smart contracts. The contracts currently do not lock pragma to a specific compiler version. Lastly, experimental feature `ABIEncoderV2` is used.

Truffle has just released a stable version of v5.0.0 of the framework which natively supports Solidity 0.5. An upgrade to this version may be worth the cost of migration, as the changes should greatly simplify project structure and testing.

Recommendation

Consider upgrading to the latest compiler version (0.5.1) as it contains several bug fixes (https://solidity.readthedocs.io/en/latest/bugs.html).

We also recommend updating Openzepellin-solidity to the final version 2.0, as along with implementing permanent interfaces for the contracts, it includes `claimOwnership` in the ownable contracts along with other subtle changes and the latest audit critical bug fixes (https://github.com/OpenZeppelin/openzeppelin-solidity/projects/2).

Closing Summary

The contracts provided for this review are of good quality.

Community audited code has been reused whenever possible. A safe math library is used to prevent overflow and underflow issues unless it is clearly not required.

No reentrancy attack vectors have been found and precautions have been taken to avoid uninitialized storage pointers that may lead to overwriting storage.

There are no transaction order issues. The token implementation is standard compliant.

The code submitted to the audit complies with general best practice guidelines. Recommended security patterns are observed throughout. Push payments are avoided in favor of pull withdrawals and the Checks-Effects-Interaction pattern.

Seven issues were found during the audit which could break the intended behavior, along with several informational notes. It is recommended for the FansUnite team to address the issues. It is furthermore recommended to post the contracts on public bounty following the audit.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of the FansUnite platform or its products. This audit does not provide a security or correctness guarantee of the audited smart contracts. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

SOLIDIFIED

Audit Report for FansIUnite January 17, 2019.