AllSporter

TGE AUDIT

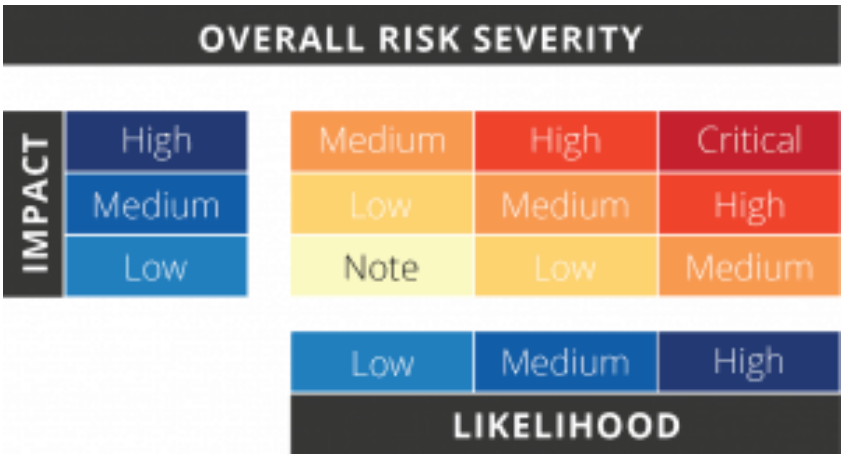
INTRO

The AllSporter team is working on a sharing economy platform dedicated to athletes. The team asked me to review their token generation contracts that will manage the process of launching the AllSporter Coin and allow the community to take a part in this process.

The code is located in the open-source github repository EthWorks/AllSporter-TGE.

The version of the code that is being reviewed was published as the commit `ab602a4f4b8163ad92f961e77f5bee181e183fbb`.

All of the issues are classified using the popular OWASP risk rating model. It estimates the severity taking into account both the likelihood of occurrence and the impact of consequences.



SUMMARY

The quality of code is very high. Although, the business logic expressed by the TGE contracts is very complex compared to a typical crowdsale, due to the well organised modular approach the implementation is clean, concise and readable. The repository contains extensive tests suits both on unit and integration level.

There have been no critical or high severity errors. Only a few medium or low severity issues have been reported. Most of the comments regard edge case scenarios and contain advice that will allow easier maintenance of the crowdsale process in all of the circumstances.

SUMMARY

Rejecting KYC may put the crowdsale into a wrong state

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

The `reject` function from the `DefferedKYC` contract decreases the amount of ether reserved in the TGE by calling the `minter.unreserve` function. This may result in a scenario when the `currentState` of the TGE is inconsistent with the amount of ether that has been collected kept in the `totalEtherContributions` variable. There is no function to rebalance the `currentState` as the state may be only increased and cannot ever move to a previous value.

One of the possible attack vectors is to attempt to buy as much tokens as possible with an account that suspicious enough to fail the KYC procedure. Although this scenario requires a large amount of ether to execute it may bring very severe consequences with corrupting the whole TGE process.

I recommend adding a function to rebalance the `currentState` by moving to the previous state with the restriction that it may be only called internally from the `unreserve` function and should also take into account the logic included in the `updateStateBasedOnTime` function.

A missing check of a mint operation result

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

The `Minter` contract invokes the `token.mint()` in its own `mint` method without checking the result. According to the `MintableToken` specification, the method returns *A boolean that indicates if the operation was successful*. Ignoring a failed minting operation can lead to unnoticed error that may propagate and corrupt the balance between the ether collected and the number of issued tokens and put at risk the whole token generation process.

I recommend checking the minting operation results by reverting a transaction if the minting operation failed: `require(token.mint(account, tokenAmount));`

UPDATE: *Fixed in commit ff24a25cc53242d61230320be70b0f54663dd8de by adding the `require` function in line 104.*

Account could be blocked for referrals before the sale is over

Severity: **MEDIUM** (Impact: MEDIUM, Likelihood: MEDIUM)

The `addFee` method in the `ReferralManager` contract marks a `referred` account so it could be used again to get a referral fee. Although, this is a very good approach to avoid an account being rewarded multiple time for the same action there is a risk that a still active account may lose a chance for a proper referral from any prospect purchases once being granted the fee .

I suggest restricting the time when a referral is being granted to a period when the crowsale is finalized and there is no chance to buy any further tokens. I will also recommend having a clear policy of combining referrals with the airdroping feature as they both depends and modify the token balance of users at the same time.

A possibility of an inconsistent TGE setup

Severity: **MEDIUM** (Impact: HIGH, Likelihood: LOW)

The `setup` function from the `Tge` contract sets the references to the key dependent contracts that take part in the token generation process. All of the dependent contracts: `Crowdsale`, `DeferredKyc`, `ReferralManager`, `Allocator` and `Airdropper` should reference back to the Tge contract as the `minter`,

However, this requirement is never checked and it may risk in creating a configuration when different dependent contracts point to different minter than expected. Having multiple minters can make it impossible to ensure that the collected ether amount is within the `saleEtherCap` limit defined in the `Tge` contract. As the `minter` reference is passed to the dependent contracts in a constructor cannot be fixed later on.

I recommend validating that the `minter` variable of all of the dependent contracts points back to the original Tge contract by adding a check: `require(_crowdsale.minter() = this)`.

UPDATE: *Fixed in commit [6d39ffd3b90580635e20064103e7d309ec8287eb](#) by adding the `ExternalMinter` contract interface and performing all of the necessary checks using the `onlyProperExternalMinters` modifier in line 65 of the `Tge` contract.*

Token setup is not fully validated

Severity: **LOW** (Impact: MEDIUM, Likelihood: LOW)

The `AllSporterCoin` is being passed to the `Tge` contract constructor as a `Crowdfundable` token. The only check that is being made is to verify that the token address is not null. As the token generation event is responsible for the creation of a new token it should be evident that the correct type of a token is being generated, the token is in a pure state and the controlling contract posses the minting privileges. Due to the fact that the token reference cannot be updated passing an incorrect variable cannot be fixed in the future.

I recommend adding additional validations to verify that the `_token` variable has correct symbol: `require(keccak256(_token.symbol()) == keccak256("ALL"))`, is in a pure state: `require(_token.totalSupply() == 0)` and is being controlled by the Tge contract `require(_token.owner() == address(this))`. To verify all of the invariant I'll advise to create a `initToken(AllSporterCoin _token)` method and ensure that the token can be set up only once.

UPDATE: *Fixed in commit [8ff5ee7a433c5a3746b05cbd3f0996897208f174](#) by adding two extra checks in lines 80-81 of the `Tge` contract.*

Unbounded loop

Severity: **LOW** (Impact: LOW, Likelihood: LOW)

The `dropMultiple` function in the `Airdropper` contract contains an unbounded loop. This loop is iterated per every item in the `accounts` array which is not limited by any constraint. Such a behaviour may lead to an unexpected `outOfGas` error and could result in a waste of resources or problems with the execution of tools responsible for the batch processing of tokens drop.

I recommend setting an explicit maximum length for the array that will prevent the execution of this method and immediately return the unused gas. It will also help to write robust scripts for contract interaction by stating a clear limit of the air drop batch that could be safely processed.

UPDATE: *Fixed in commit [3dfd388ea8ae4b7ff580a67b0370854c38426df4](#) and updated in commit [1af930c489c73c1b32c4d8979bb08d72ce83c956](#) by introducing the `MAXIMUM_LOOP_BOUND = 15` parameter which limits the number of accounts that could be processed in a single batch.*

Outdated version of the Open Zeppelin library

Severity: **LOW** (Impact: MEDIUM, Likelihood: LOW)

The `OpenZeppelin` library version referenced in the `package.json` file is outdated as it points to the `1.6.0` release which was published in January 2018. First, the old version is not compatible with the the current solidity compiler version `0.4.24` that is used in the project. Moreover using an outdated library brings a risk of missing all of the recent security updates and patches.

I recommend updating the `OpenZeppelin` library version to the most up to date one which is `1.12.0`, by changing the `package.json` to include: `"openzeppelin-solidity": "1.12.0"` (beware the package name was also updated) and rebuilding the project.

UPDATE: *Fixed in commit [1af930c489c73c1b32c4d8979bb08d72ce83c956](#) by updating the `openzeppelin-solidity` library to the version `1.12.0`.*

Anyone can release tokens

Severity: **LOW** (Impact: LOW, Likelihood: LOW)

The `SingleLockingContract` correctly restricts access to the `releaseTokens` function only to the contract owner or tokens beneficiary. However, the `Allocator` contract that owns the locking contracts opens the access to the `releaseLocked` function to anyone. This effectively removes all of the access constraints of the `SingleLockingContract`, so that the moment when tokens are transferred is beyond the control of the beneficiary or the TGE operator.

I recommend restricting the access to the `releaseLocked` function only to the contract owner or the tokens beneficiary so it's consistent with the logic contained in the `SingleLockingContract`. This could be implemented by adding the check: `require(msg.sender == account || msg.sender == owner);`

UPDATE: *Fixed in commit [7c4d459054ffb8b897d60b8ac4f1fc5682d409d4](#) by enforcing that the `releaseVested` and the `releaseLocked` functions can be invoked only by the owner of the `Allocator` contract or the token beneficiary.*

Treasury address cannot be updated

Severity: **LOW** (Impact: LOW, Likelihood: LOW)

The `Minter` contract sets the `treasury` address in its constructor. The target cannot be updated later on. Usually, the whole token generation event is a long

process and it's best to allow the operators maximum flexibility to cater for an unpredicted scenarios, like attempts to hack the treasury contract when all of the ether is kept.

I suggest granting the `Minter` contract owner a privilege to update the treasury contract as an extra safety measure to secure the funds raised during the crowdsale by adding a function `updateTreasury(address _treasury)external public onlyOwner;`

UPDATE: *Fixed in commit [12c6ea1030a3e5ef8f2c3c64cb3abe0ae98a36fd](#) and commit [a036c1fb9792fe63282998f339f7b47def905688](#) by adding the* `updateTreasury` function in the `Crowdsale` *contract which forwards the call to the* `DeferredKYC` *and sets the new treasury address.*

Allocator doesn't verify token percentages

Severity: **LOW** (Impact: MEDIUM, Likelihood: LOW)

The `Allocator` function contains the `validPercentage` modifier which is never used in the contract to validate the correctness of token distribution. Moreover, the percentages should sum up to 100 which is also never checked. This brings a risk that any last minute update of the percentages could be overlooked and lead to a corrupted TGE.

I recommend verifying the correctness of configuration in the constructor of The `Allocator` contract by checking for every constant `validPercentage(CONSTANT)` and making sure that the sum is equal to 100.

UPDATE: *Fixed in commit [6793ca5f97546a1692a9f90ae65e3fe7b798ccbb](#) by applying the* `validPercentage` modifier to all of the allocations and checking if they sum up to 100.

States and global caps may be inconsistent

Severity: **LOW** (Impact: MEDIUM, Likelihood: LOW)

The `Tge` contract have two different ways of limiting the contributions. The first one uses a global `_saleEtherCap` variable and the second one relies on defining a limit per a single state by passing a `singleStateEtherCap` variable in the `setup` function. However, there is no verification to check if the two limits are consistent, so there is a risk of setting a `singleStateEtherCap` that is big enough to effectively disable higher states of the crowdsale.

I recommend verifying the consistency of the limits by adding a precondition `require(singleStateEtherCap.mul(8) < saleEtherCap)` in the `Tge` contract `setup` function.

UPDATE: *Fixed in commit [3d28839b4747c50fc58bba19c966405a3e58fa85](#) and [7ef9f90d1aec2da39451f2d35a3c7bf78f4fb255](#) by checking the relations between* `singleStateEtherCap` *and* `saleEtherCap` *in line 130.*

NOTES AND SUGGESTIONS

- Currently, there is no mechanism to limit the number of tokens that is bought by a single person. This may result in a single investor overtaking a control over the token and dominate the project. I recommend putting a limitation which may be effective especially as there is a KYC mechanism in place in the form `DefferedKYC` which prevent the possible workaround by account splitting.

- The code that performs validation e.g. `onlyValidAddress` directive is repeated in multiple contracts. This not only makes the code less readable but brings a risk of introducing errors when the modifier needs to be updated. I recommend extracting the common logic into an abstract base contract.

- The Bought(msg.sender, msg.value); emitted in the `Crowdsale` contract may be misleading to investors or anyone who watches the TGE process, as it indicates that the token purchase operation has been finalised. In reality, this indicates only the beginning of the purchase process as the buy intention needs to be approved first by the `DefferedKYC` contract. I recommend updating the name to reflect the true meaning of this action.

- The `SafeMath` library referenced in the `SingleLockingContract` contract is never used. I recommend removing it as it pollutes the code and makes the deployment procedure more complex and expensive. ***UPDATE:*** *Fixed in commit 7ef9f90d1aec2da39451f2d35a3c7bf78f4fb255.*

- I recommend defining valued that are used more than a once as constants to avoid errors during an update or refactor of code. One example of such a value is `18` which is used to denote the number of digits in the token decimal representation in the `AllsporterCoin` contract.

- Consider adding a helper method `canBuy()` in the `Crowdsale` contract so the users can easily check if the sale is open before transferring the funds. This will not only make your contracts API more user-friendly but also may help to investigate situations when a transaction has been reverted.

- I recommend logging vesting contract addresses in the functions `allocateCustomer` and `allocateTeam` of the `Allocator` contract to improve tracking the locked tokens location. ***UPDATE:*** *Fixed in commit 7ef9f90d1aec2da39451f2d35a3c7bf78f4fb255.*

Jakub Wojciechowski
Smart contract auditor