Hubii Token Audit

OPENZEPPELIN SECURITY | AUGUST 11, 2017

Security Audits

The Hubii team asked us to review and audit their crowdsale and “Hubiits” (HUB) token contracts. We looked at the code and now publish our results.

The audited contracts can be found in the CoinFabrik/ico repository. The version used for this report is commit `142f43f3bec4afd0ffff3fce06946cf67cb0272f`.

We’ve found the code to be very modular and well documented. Here’s our assessment and recommendations, in order of importance.

Update: Hubii team followed most of our recommendations in the latest version of the code.

Severe

State transitions are implicit and not enforced

There are multiple states a `Crowdsale` instance can be in. The current state can be obtained by calling `getState`, and functions are enabled or disabled according to it. Said function also gives an idea of the desired sequence of states that the crowdsale should follow. It’s easy to see, however, that there are a lot more possible state transitions than those implied:

- If at any point in time the finalizeAgent variable is changed into a non-sane instance, a crowdsale can return to the `Preparing` state.
- Once the crowdsale has been considered successful (being in the `Success` state), it’s possible to resume the `Funding` state by setting a new cap.

and enforcing the states in which each function can be called.

Some examples of explicit state transitions correctly implemented are OpenZeppelin's RefundVault and Solidity's Documentation's StateMachine.

Update: Hubii replied *"The function where the state transitions were possible were marked as `onlyOwner`, lowering the possibility of outside attack. Despite the function having the `onlyOwner` modifier, we modified the contracts to prevent unexpected state transitions and removed the `Preparing` state."* Fixed in `5a90957abf3f792951b3aad66d6eb0317f0500ca`.

Incorrect investor count

Calls to `preallocate` in Crowdsale.sol do not increase the `investorCount` variable. If the intention is to count investors who transfer funds using this function, consider increasing the variable in this method.

Update: Hubii replied that the meaning of this variable was by design. Hubii replied *"The `investorCount` was designed to count investors in the crowdsale, and not early investors assigned through the preallocate function. It is designed in this way."*

Furthermore, the check in line 191 for `investedAmountOf` will not increase the investor count for regular purchases if the calculated price is zero. Consider changing the guard to check for `tokenAmountOf` instead.

Update: Hubii replied *"Regular purchases are required to send non-zero amounts and the `investorCount` is only increased when `investedAmountOf` is zero to avoid counting the same address twice. Early investors who also purchased tokens through the crowdsale will also be counted correctly by checking `investedAmountOf`, which won't happen if we check `tokenAmountOf`."*

Inconsistent order of arguments of calculatePrice

The function `calculatePrice` is defined in `PricingStrategy` to take five arguments including `weiRaised` and `tokensSold`. These two are swapped in the implementation of the function in `FlatPricing` and its use in `Crowdsale`. It presently causes no problem because `msgSender`, `uint decimals)`.

Update: Hubii replied *"The name of the arguments and the documentation was fixed to reflect the actual usage. But, this issue is not severe since it does not affect the contract behavior."* Fixed in `c3fc1798313e3c2bd98decb57f97f74e5f003209` .

Token contract creator has control over token emission and release

Instead of the `Crowdsale` acting as owner of the token, as in most implementations (see OpenZeppelin's `Crowdsale`, there is an account that retains ownership of the token. This allows the owners to appoint themselves as minting agents at any point during the crowdsale and emit new tokens, or release the token before the crowdsale ends. This reduces the public trust on the crowdsale mechanics. Move ownership of the token to the `Crowdsale` contract, and maintain control only over the upgrade mechanism by appointing the team as `upgradeMaster`.

Update: Hubii replied *"As suggested in another issue the setup was changed to create all the contracts from a single contract. In this way, the `Crowdsale` contract is the owner of the `CrowdsaleToken`."* Fixed in `e9e05d7adc9efde724afee0facfc89e76db67efe`.

Strategies and limits can be modified mid-crowdsale

Methods `setEndsAt`, `setPricingStrategy`, `setCeilingStrategy`, `setFinalizeAgent` and `setFundingCap` in Crowdsale.sol allow the owner to change the crowdsale rules while the crowdsale is running, changing when the crowdsale ends, the funds cap, the price and the maximum investment per address, at any stage. This requires complete trust on the owner of the crowdsale. Consider requiring these parameters at construction time, or restricting these changes to the `Preparing` stage of the crowdsale only.

Update: Hubii replied *"These methods are declared internal to allow modifications only during the deployment of the `Crowdsale` contract, and thus, the subcontracts cannot be changed afterwards."* Fixed in `5a90957abf3f792951b3aad66d6eb0317f0500ca`.

Preallocate function has no sanity checks

(allowing preallocations to be sent even on a [illegible] or [illegible] crowdsale), or the cap not being exceeded. Even if the method is to be invoked only by the `Crowdsale` owner, consider adding checks for all these conditions to prevent inconsistent states produced by human errors.

Update: Hubii replied *"Checks were added to ensure correct values are processed by the `preallocate` function to avoid any problems."* Fixed in `b4fccc6dee752f9b64b5598270032fd9cec106f6`.

Potential Problems

Contracts are unnecessarily generic and complex

Both the `Crowdsale` and `CrowdsaleToken` contracts depend on several flags to control optional behaviors. Some examples are:

- The crowdsale contract can optionally have a `minimumFundingGoal`, which can trigger a `Failure` state if set.
- The token has a `mintable` flag that controls whether it can actually mint, even though it extends `Mintable`.

The many different flags and available behaviors interact in many ways, resulting in a very complex contract. More complexity means a larger attack surface. Rather than conditionally controlling different behaviors via flags, consider settling on a single behavior per class to reduce complexity.

Update: Hubii replied *"Some complexity is necessary if your contract has to provide several options. We think that altering significantly the contracts is too risky at this stage of development."*

Contract initialization is too complex

The excessive use of the strategy pattern for the crowdsale contract makes deployment and configuration much more complicated, requiring several agent configurations (`setMintAgent`, `setReleaseAgent`, `setFinalizeAgent`), which is highly error prone. Consider defining a specialized version of `Crowdsale` (maybe calling it `HubiiCrowdsale`) that performs all required initialization steps, including setting up and configuring the agents and token contracts.

the [illegible] modifier prevents similar attacks.

Update: Hubii replied *"The split of the contract in several pieces was done to circumvent the gas limit in the mainnet last year. We understand this is not an issue now, and we have grouped the setup process within a single contract."* Fixed in `5a90957abf3f792951b3aad66d6eb0317f0500ca`.

Owner can load more refund money than necessary

`loadRefund` allows to send balance into the contract so that it can be used for refunds. This money can only be withdrawn via the `refund` function, by an investor requesting a refund of the same amount they had invested.

It's possible to load more balance than necessary, and the contract provides no mechanism to withdraw the extra balance. (Note that it would be possible to withdraw it by abusing the lack of safety checks in `preallocate`, as explained below, but this should not be relied upon.) Consider requiring that `msg.value == weiRaised` in `loadRefund`.

Update: Hubii replied *"We decided to assign the excess of the refund ether to the sender address. So they can refund the excess through the normal mechanism."* Fixed in `b6f8f29e8e6578d9efc56457c0aa8ff3a49f2c05`.

Unclear whether a finalize agent is required for a crowdsale

Although the code of the finalize function points out that *"Finalizing is optional. We only call it if we are given a finalizing agent"*, the `Crowdsale` can never move out of the `Preparing` state as per L481 if no finalize agent is set.

If the intention is for the finalize agent to be mandatory, consider accepting it as part of the constructor parameters, and remove the method to change it on the fly. On the other hand, if the intention is to support no finalize agent, ensure there is an alternative way to release the tokens, which is currently responsibility of the finalize agent.

Update: Hubii replied *"As part of another issue the `Preparing` stage was removed, and now the `finalizeAgent` is mandatory."*

Fund receiving address can be changed by owner for a brief period

The purpose stated in the comments is to be able to fix it before the crowdsale has begun, but this is not enforced by the contract.

If the crowdsale is already running, with less than 5 investors, the owner can still change the address. If by error or malice he sets an invalid one, soon afterwards it will be impossible to change, once investor count reaches 5. For this reason, it's not a good safety measure as it stands. Consider removing the function, or adding the `onlyInEmergency` modifier.

Update: Hubii replied *"The function to change the receiving address was declared `internal` to only allow modifications when the contract is being deployed."* Fixed in `5a90957abf3f792951b3aad66d6eb0317f0500ca`.

Ownable's transferOwnership fails silently

In `transferOwnership` there's a sanity check to see that the address argument is non-zero, which can fail silently. Consider throwing if the check doesn't pass, by changing the line to `require (newOwner != 0x0);`.

Update: Hubii replied *"The `transferOwnership` function was modified to require the `newOwner` to be non null."* Fixed in `5a90957abf3f792951b3aad66d6eb0317f0500ca`.

Use SafeMath in all math operations

Most math operations are safe, but there's still a few that are unchecked (these two, for instance). It's always better to be safe and perform checked operations.

Update: Hubii replied *"We updated to use safe math operations where they were not used."*

Strategy and agent contracts can be set to invalid ones

Strategy contracts `PricingStrategy` and `CeilingStrategy` as well as `FinalizationAgent` each have an `isSane` function and an accompanying one in `Crowdsale`, whose apparent purpose is to validate their correct setup with respect to a `Crowdsale` instance. Although the instances can be changed mid-crowdsale, they are not validated, and the contract could be left in an invalid state. Consider adding the validation for each in the setters `setPricingStrategy`, `setCeilingStrategy` and `setFinalizeAgent`. For example, add `isPricingSane();` after line 396.

Warnings

Solidity version

Current code specifies version pragma `^0.4.11`. We recommend changing the solidity version pragma to the latest version (`pragma solidity ^0.4.14`) to enforce the use of an up to date compiler; especially considering the latest version includes a security fix for the `ecrecover` function, which Crowdsale.sol uses.

Update: Hubii replied *"We updated our requirements to solidity v0.4.13, and removed the use of `ecrecover` since it was not essential to the `Crowdsale`. We didn't update to v0.4.14 because it isn't supported on Truffle yet."*

Compiler warnings

The compiler issues the warning *Variable is declared as a storage pointer. Use an explicit "storage" keyword to silence this warning* in MultiSigWallet.sol. Consider adding the required keyword to remove this warning.

Update: Hubii replied *"We have added the keyword storage as suggested."* Fixed in `5a90957abf3f792951b3aad66d6eb0317f0500ca`.

ERC20 decimals field should be uint8

The decimals field of the token is defined as `uint256` both in `CrowdsaleToken` and `FractionalERC20`. It is, however, defined as `uint8` in the ERC20 standard. Consider changing the type to `uint8`.

Update: Hubii replied *"We have updated the contracts to reflect the standard and declared decimals as `uint8`."* Fixed in `312d8be8002dc99074dbd2a0a6db1075cc5623ed`.

Undocumented interface of UpgradeAgent

Compared to the rest of the code, `UpgradeAgent` is lacking in documentation and clarity. Since this is intended to be used by future developers, we believe it's specially important that it be very well documented.

parameters to transfer from the older to the new token. This function is called by the older token.

Consider renaming the parameters to owner and amount, which are more representative of their meaning. Also, consider enforcing in `UpgradeAgent` that `upgradeFrom` be only callable from the old token.

Update: Hubii replied *"We have added documentation to the `UpgradeAgent` to help understand its functionality."* Fixed in `a042810f569192b074f22f33df3b52c6cdafe53a`.

Low test coverage

There is a single test file in the codebase, which only checks a few methods on the `Crowdsale` contract. Consider adding tests for finalization, refunds, crowdsale failures, preallocation, investment with customer ID and signature, among others; as well as tests for other contracts, such as the `CrowdsaleToken`, including tests for the releasing and upgrading mechanisms.

Also, all tests are executed on the same set of deployed instances. This restricts all tests to be run in succession and on a single configuration, limiting testing to a single set of constructor parameters for all contracts. Consider setting up different instances on different test suites to accommodate for different configurations, as well as adding unit tests for all contracts using mocks where needed.

Update: Hubii replied *"We have added several tests since the audit, and we are adding more."*

Does not install libraries via a package manager

Though the code uses OpenZeppelin's `BasicToken`, `ERC20`, `ERC20Basic`, `Ownable`, and `SafeMath`, code is copied and pasted into the repository, including some style modifications such as renaming `uint256` to `uint`. This makes it difficult to include any security changes included with new releases of the library, as they must be manually integrated. Consider using npm for installing OpenZeppelin.

Update: Hubii replied *"We will consider this for future changes in the repository."*

Token for Crowdsale must be Mintable

Consider changing the token field type to `MintableToken`.

Also, if the `CrowdsaleToken` contract is used as a token for the `Crowdsale`, the token's flag `mintable` in the constructor must be set to true for the crowdsale to work. Consider removing this flag altogether, and always making `mintable == true`.

Update: Hubii replied *"The `Crowdsale` contract was changed to always expect a `Crowdsale` token. We require the `CrowdsaleToken` to be mintable in this instance of the `Crowdsale`."*

Refunding is not trustless

The crowdsale has logic to return the funds to original investors if the minimum cap is not reached (seeL437). However, this depends on the team from manually transferring the funds back to the `Crowdsale` via the `loadRefund` method, requiring the public to trust in a third party for effectively issuing the reimbursements. Consider locking the funds in a contract specifically designed for it until the crowdsale is known to have failed or succeeded, and only in the latter case transfer them to the team.

Update: Hubii replied *"The audit is correct to point that the refunding in case of the crowdsale not reaching the minimum cap is not trustless. We consider that fixing this issue at this point of the schedule will add more risks than solutions. But we will consider it for future reference."*

Timestamp usage

There's a problem with using timestamps and `now` (alias for `block.timestamp`) for contract logic, based on the fact that miners can perform some manipulation. In general, it's better not to rely on timestamps for contract logic. The solutions is to use block.number instead, and approximate dates with expected block heights and time periods with expected block amounts.

The Crowdsale.sol contract uses timestamps in lines 484–485. The risk of miner manipulation, though, is really low. The potential damage is also limited: miners could only slightly manipulate when the crowdsale starts or ends. We recommend the team to consider the potential risk of this manipulation and switch to `block.number` if necessary.

For more info on this topic, see this stack exchange question.

Notes and Additional Information

- Good job fixing warnings and typos from Consensys MultiSig wallet! Consider sending a pull request to the official repository so these changes are integrated back into main development.
- Consider making `CrowdsaleToken` inherit from `FractionalERC20`.
- In order to improve clarity, consider moving precondition checks in methods to modifiers in all methods in BonusFinalizeAgent.sol and UpgradeableToken.sol, as well as in investment methods, `setPricingStrategy`, `setCeilingStrategy`, `setEndsAt` and `loadRefund` in Crowdsale.sol.
- Consider reusing the `setCeilingStrategy` function in place of the two equivalent lines in the `Crowdsale` constructor.
- `Crowdsale` will accept dates in the past as parameters for start and end. Consider adding sanity checks for that.
- Prefer expressing rate as `tokensPerWei` instead of `weiPerToken` in FlatPricing.sol, so the amount of tokens to be issued per investment can be calculated as a multiplication instead of a division, avoiding rounding errors.
- Consider including the unit in which the bonus is expressed as part of the variable name to avoid confusions in `BonusFinalizeAgent` L29.
- Function `isSane` in line 21 of PricingStrategy.sol could be made abstract. This would make it harder for a developer implementing a child contract to forget implementing the method.
- Comment in line 54 of MintableToken.sol is misleading. "Only crowdsale contracts are allowed to mint new tokens." Any address can be a mint agent.
- `Crowdsale` contract is `Haltable` but this is only used by adding the modifier `stopInEmergency` to functions `investInternal` and `finalize`. Consider adding that modifier to other independent state-modifying functions such as `loadRefund` or `refund`.
- The comment in line 80 of UpgradeableToken.sol is incomplete.
- Consider renaming `transferAgents` in `ReleasableToken` to something more descriptive, such as `lockExemptions`.
- Comment in line 47 of Crowdsale.sol seems to be wrong: reads "tokens will be transfered from this address" but should be "ether will be transferred to this address"

Consider extracting the [illegible] [illegible] mechanism and [illegible] method into `Refundable` and `Preallocateable` contracts respectively, to improve modularity.
- Consider breaking BonusFinalizeAgent.sol into two contracts: a base one with the `isSane` check (if needed) and the `releaseTokenTransfer` call upon finalization, and a derived one with the bonus logic itself.
- The check for `require(!finalized)` in `finalize` in L317 of Crowdsale.sol is redundant, since the method can only be called in state Success as per its modifier.
- Function `investInternal` in line 171 of Crowdsale.sol and `preallocate` in line 239 of Crowdsale.sol share a lot of code and could be refactored to using a common function.
- `getState` is called twice in the same function in `investInternal`. Consider saving the state to a variable to avoid the double call. This will also save gas.
- Consider changing all `assert(send)` calls in Crowdsale.sol (L207, L212 and L443) to `transfer` for simplicity.
- Consider extracting the magic constant `10000` from L56 of `BonusFinalizeAgent` into a constant state variable, for clarity and maintainability.

Conclusion

We’ve found the code to be very modular and well documented. Six severe security issues were found and reported, along with how to fix them. Some changes were proposed to follow best practices and reduce potential attack surface.

Update: Hubii team clarified that some of the issues found were design decisions rather than vulnerabilities. The team fixed most of the issues following our recommendations.

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the HUB token and crowdsale contracts. We have not reviewed the related Hubii project. The above should not be construed as investment advice or an offering of HUB tokens. For general information about smart contract security, check out our thoughts here.

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits