UnikoinGold Token Audit

OPENZEPPELIN SECURITY | NOVEMBER 7, 2017

Security Audits

The CoinCircle team asked us to review and audit their UnikoinGold token and distribution contracts. We looked at the code and now publish our results.

The audited contract is in the unikoingold/UnikoinGold-UKG-Contract Github repository. The version used for this report is the commit `8b38f30039c2d13383c416fd6143f6bd0f091404`.

Good job using OpenZeppelin to write minimal extra code. Excellent automated tests code coverage, with 96% for `TokenDistribution` and 95% for `ParticipantAdditionProxy`.

Here's our assessment and recommendations, in order of importance.

Critical Severity

No critical severity issues were found.

High Severity

Unnecessary claim step for sale contributors

Contributors participate in the sale by transferring their balance to a set of trusted addresses managed by Unikrn, who then distribute the tokens in a later step via the `TokenDistribution` contract. Token allocations are set for each contributor, who must then claim the tokens to have them added to their balance.

allocated to their addresses. This way, there will be indeed 1B tokens, even if some participants cannot move them afterwards.

Furthermore, if a participant did misplace their keys, tokens are locked in the TokenDistribution contract (as per L136) and there are not methods to take them out of the contract, so the tokens are created but are unusable. A participant may misplace their keys after the distribution, so it makes no sense to try to prevent allocation to addresses with lost keys.

Consider removing the claim mechanic altogether, and allocate the tokens to the contributors directly. The whole `ParticipantAdditionProxy` contract seems to be unnecessary. This reduces the complexity of the distribution process, simplifies the code, and reduces the hassle for end users to access their purchased tokens.

Update: *The CoinCircle team indicated that this was motivated by the fact that the proxy contract will be deployed before the distribution contract, in order to store the future owners of the token before the final implementation is finalized:*

> The design decision for the contract was to separate concerns: the proxy contract will be deployed prior to the distribution contract. The proxy contract will be populated by Unikrn, and not be made known to the public until the contract is verified, locked and unchangeable

Check order of timestamps in Token distribution constructor

The TokenDistribution contract relies on three timestamps:

- `freezeTimestamp`, which indicates up to which point it is possible to cancel the distribution contract
- `distributionStartTimestamp`, that signals when the distribution starts
- `lockupTimestamp`, which indicates when locked tokens may be released

Though it is assumed that `freezeTimestamp` should be before the `distributionStartTimestamp`, this is not checked in the code. If this precondition is not respected, an owner may stop the contract mid-distribution.

Also, for the lockup to have any effect, it is necessary for `lockupTimestamp` to occur after the `distributionStartTimestamp`. Otherwise, locked tokens can be claimed as soon as

`distributionStartTimestamp`, and another for `distributionStartTimestamp < lockupTimestamp`, in the TokenDistribution constructor.

Update: *First guard added in this commit. Second one is unnecessary as per this commit.*

Make Token ERC20 Compliant

There's several ways in which the UKG token is not ERC20 compliant.

- The `totalSupply` variable is not updated correctly. It should always contain the total token supply. For example, token balance is given to the Unikrn team in line 138 of TokenDistribution.sol and `totalSupply` is left unchanged. Consider updating `totalSupply` every time tokens are created or destroyed.

Update: *Fixed in this commit.*

- Consider adding public `name`, `symbol` and `decimal` properties to the Token contract. Though not mandatory, they are recommended as part of the ERC20 specification. See OpenZeppelin's SimpleToken as an example.

Update: *Fixed in this and this commits. Remember also to bump `version` field to 1.0 and mark it as constant once the final code is frozen.*

- Consider emitting Transfer events from the 0x0 address when creating new tokens, instead of (or additionally to) emitting custom CreateUKGEvent events in the `TokenDistribution` constructor L137 and L139.

Update: *Fixed in this commit.*

Consider making these changes to make the token respect the ERC20 standard.

Medium Severity

Split Token and Distribution into different contracts

The token Distribution process is built into the same Token contract in TokenDistribution.sol. Since the distribution is not part of the token intrinsic mechanics, it is strongly recommended to use a

This simplifies the logic of the token contract, separates different concerns among different contracts, and also reduces the attack surface for the token contract.

Update: *Fixed in this, this, this and this commits.*

Use SafeMath in all math operations

Though some math operations are safe, there are others that are unchecked in TokenDistribution lines 162, 236, 294. It's always better to be safe and perform checked operations.

In particular, line 235 subtracts the current phase allocation from the sender's remaining allowance, *without checking that the minuend is greater or equal than the subtrahend*. Even though the distribution logic should not allow for this to happen, it's highly recommended to always check that a user has enough balance when subtracting from the allowance, which is done automatically when using SafeMath.

Update: *Fixed in this commit.*

Fail early and loudly

In the spirit of failing as promptly as possible in all methods, consider adding checks in functions `allocatePresaleBalances`, `allocateSaleBalances`, and `allocateLockedBalances` of **ParticipantAdditionProxy** to ensure that both arrays passed in as parameters have the same length. If the first array passed in is shorter than the second by mistake (ie if an address is missing for an allocation), then the code will silently continue.

Update: *Fixed in this commit.*

Low Severity

Install OpenZeppelin via npm

Code from `SafeMath`, `Ownable`, `ERC20Basic`, `ERC20`, `BasicToken`, and `StandardToken` was copied from OpenZeppelin into the files SafeMath, Ownable, and Token.

Consider following the recommended way to use OpenZeppelin contracts, which is via the zeppelin-solidity npm package. This allows for any bugfixes to be easily integrated into the codebase, such as issues 375 and 400 which are fixed in the latest release and affect the Token contract.

Update: *Fixed in this commit.*

Redundant state variables in ParticipantAdditionProxy

For both the sale and presale participants, the ParticipantAdditionProxy keeps track of whether the entire addition process is complete (`presaleAdditionDone`, `lockedAdditionDone`, and `saleAdditionDone`), which is the balance for each address(`presaleBalances`, `lockedBalances`, and `saleBalances`), and whether an address has already been allocated (`presaleParticipantAllocated`, `lockedParticipantAllocated`, and `saleParticipantAllocated`).

Assuming that each participant has a non-negative allocation, and since once a participant is allocated it cannot be modified, then the flag stating whether a participant has been allocated can be replaced by a check of whether the balance for the address is zero or not. This allows the three `ParticipantAllocated` variables to be removed.

Furthermore, since it is not possible to allocate balance over the pre-defined cap, and the process can only be marked as complete when the cap is reached, the flags and methods for ending the addition process (`presaleAdditionDone` / `lockedAdditionDone` / `saleAdditionDone` and `endPresaleParticipantAddition` / `endLockedParticipantAddition` / `endSaleParticipantAddition`) are not needed, and can be replaced by a check on whether the cap was reached.

Removing redundant state variables not only reduces gas costs due to reduced storage used, but also greatly simplifies the code. Consider doing so.

Solidity version

Current code specifies version `pragma ^0.4.11`. We recommend changing the solidity version pragma to the latest version (`pragma solidity ^0.4.17`) to enforce the use of an up to date compiler.

Update: *Updated to 0.4.15 in this commit, latest version of solidity supported by the used Truffle version.*

Confusing use of unneeded temporal variables

Throughout the code, in many functions a **temp** variable with a calculation is defined, an assertion is performed over the temp value, and if it succeeds, then a state variable is assigned with the value `temp`. See as an example ParticipantAdditionProxy L70:

```
`uint256 tempPresaleTotalSupply =
presaleAllocationTokenCount.add(approvedPresaleParticipantsAllo
`require(tempPresaleTotalSupply <= PRESALE_TOKEN_ALLOCATION_CAP
`presaleAllocationTokenCount = tempPresaleTotalSupply;`
```

The same happens in ParticipantAdditionProxy L92 and L114, and TokenDistribution L158 and L290.

Since a failed assertion via **require** will revert all state changes within the transaction, it is not necessary to use this temp variable. Assigning directly to the state variable and then checking the assertion will have the same effect, and make the code shorter and easier to read. Consider removing these unneeded temporal variables.

Update: *Fixed in this and this commits.*

Usage of magic constants

The total number of phases in the TokenDistribution is repeated as a magic constant throughout the contract code, in L142, L189, L214, L215 and L239.

Update: *Fixed all except for L142 in this and this commits.*

Unused state variable phasesClaimable

State variable `phasesClaimable` in `TokenDistribution` is unused. Moreover, there is a function with the same name in L198 which shadows the getter generated by the `public` modifier, and has entirely different semantics. Consider removing the `phasesClaimable` field.

Update: *Fixed in this commit.*

Use of integer type to represent boolean

State variable `isVesting` in `TokenDistribution` is defined as a map from addresses into `uint256`, though only 0 and 1 values are used as a proxy to represent boolean values. Futhermore, 0 is used to represent `true` and 1 is used to represent `false`, which is the opposite as to how boolean values are represented in the ABI, . Consider changing the type of the map values from `uint256` to `bool`, or removing the field altogether, as explained below.

Update: Fixed by removing the field (as suggested below) in this commit._

Redundant state variable isVesting

State variable `isVesting` in `TokenDistribution` is set to true for an address if and only if a participant reached the last phase in `claimPresaleTokens`. As such, it is straightforward to infer whether a participant is vesting or not just by checking if `phasesClaimed` for the address is equal to 10. Consider removing the redundant state variable (`isVesting`) if it will not have any other uses.

Update: *Fixed in this commit.*

Redundant state variables claimed and phasesClaimed

State variables `claimed` and `phasesClaimed` in `TokenDistribution` hold the same information: whether an address has claimed a particular phase for the presale. The former keeps track of both through a mapping `phase =&gt; address =&gt; isClaimed`, while the latter stores the last phase claimed for a given address. Given that phases can only be claimed

```
`claimed[aPhase][anAddress] == (phasesClaimed[anAddress] >= aPh
```

Consider removing either of the two (we suggest removing `claimed`) in favour of the other to remove redundant state from the contract.

Update: *Both variables were decided to be kept to reduce complexity of the `phasesClaimable` function implementation.*

Notes

- Some constants are redefined in different locations. Example: `PRESALE_TOKEN_ALLOCATION_CAP` is defined in both TokenDistribution and ParticipantAdditionProxy. (**Update**: *kept for readability).*
- Creating modifiers that are only used once is more confusing than useful. For example, `presaleTokensStillAvailable` defined in L102 of TokenDistribution.sol is only used in L255 of the same file. Consider having that as a regular function precondition. (***Update**: fixed in this commit).*
- Consider removing redundant getter functions for fields already marked as `public`. Solidity automatically creates getter functions for these fields, so it is not necessary to redefine functions to access them from outside the contract. Examples of these functions are `balanceOfPresaleParticipants` , `balanceOfLockedParticipants` and `balanceOfSaleParticipants` in ParticipantAdditionProxy; as well as all the test functions (`presaleParticipantAllowedAllocationTest` , `allocationPerPhaseTest` , `remainingAllowanceTest` , `saleParticipantCollectedTest` , and `isVestingTest`) defined in TokenDistribution L314–L332. (***Update**: fixed in thesetwo commits).*
- Consider adding a check for phase being strictly greater than zero in claimPresaleTokensIterate, even though the function is internal and only invoked with values greater than zero, for additional security in case the calling code is changed. (***Update**: fixed in this commit).*
- The function `balanceOf` from BasicToken is repeated in TokenDistribution L310, which inherits from BasicToken. Consider removing the redundant code. (***Update**: fixed in this*

abstracted, to reduce code repetition. See for example `allocateSaleBalances`, `allocateLockedBalances`, and `allocatePresaleBalances`, which are almost exact copies of the same code, with different variable names. The same happens with `claimLockedTokens` and `claimSaleTokens` in TokenDistribution. (***Update**: locked logic was removed in this commit, reducing the number of copies of the same code*)._

- For-loops found in code are bounded and safe._
- The function `min` in L180 of `TokenDistribution` is already implemented in the Math library of OpenZeppelin. Consider reusing the available function from OpenZeppelin rather than copying it into the contract. (***Update**: fixed in this commit).*
- Consider adding a comment to the field `claimed` in L77 of `TokenDistribution` to clarify that the mapping is indexed by the presale phase. (***Update**: fixed in this commit).*
- Consider moving the `cancelDistribution` flag, along with the modifier and method to set it, to a separate `Cancelable` contract, similar to OpenZeppelin's Pausable contract, to improve separation of concerns._
- The first iteration of the loop for computing `endOfPhaseTimestamp` in L142 of `TokenDistribution` is skipped and executed manually outside the loop. Consider executing it within the loop to reduce code duplication. (***Update**: fixed in this commit).*
- As an alternative to the previous item, consider removing the state variable `endOfPhaseTimestamp` altogether in order to reduce gas costs from unnecessary storage, since this value can be easily calculated on the fly whenever needed._
- Comments on state variables numLockedTokensDistributed and lockupTimestamp are copied from the respective previous lines. Consider updating them to properly document the variables. (***Update**: fixed in this commit).*

Update 51fcf759

After the original audit, the Unikrn team asked us to review two additional changes:

- The possibility to correct mistakes when setting up the participants list (commit fcd4b0ce).
- A change from pull to push transfers in the tokens distribution, so end-users do not need to pay the gas cost for claiming their tokens from the sale (commit 51fcf759).

- Great job on automated tests, with coverage reaching an impressive 99%.
- With the addition of the `removeParticipant` method, it is now possible to remove or change the number of tokens for a participant, as long as the sale is not marked as done via the `saleAdditionDone` flag. In order to prevent any modifications *during* the distribution, consider adding a check in `distributeSaleTokens` to ensure that `saleAdditionDone` is set in the `participantData ProxyContract`. This prevents a distribution from taking place using a `ParticipantAdditionProxy` that might still change. Consider doing the same for the `claimPresaleTokens` method and the `presaleAdditionDone` flag.
- Methods `balanceOfPresaleParticipants` and `balanceOfSaleParticipants` in the `ProxyContract` interface L32–33 are not needed and can be removed to prevent confusions, since they are unimplemented by the `ParticipantAdditionProxy` contract.

Conclusion

No critical severity issues were found. Some changes were proposed to follow best practices and reduce potential attack surface.

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

Related Posts

Beefy

BRUSHFAM
OpenBrush Contracts

Linea

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