Open in app Get started

Published in New Alchemy

New Alchemy Follow
May 22, 2018 · 9 min read · Listen

Save

The Rouge Project Smart Contract Audit

The Rouge Project engaged New Alchemy to audit the smart contracts for the RGE token and the associated crowd sale. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts, finding differences between the contracts' implementation and their behavior as described in public documentation, and discovering any other issues with the contracts that may impact their trustworthiness.

The Rouge Project provided New Alchemy with access to the relevant source code and whitepapers. An initial version of this document was provided to the Rouge Project, who made various changes to their contracts based on New Alchemy's findings; this document was subsequently updated to reflect the changes.

New Alchemy's audit was additionally guided by the Rouge Network website[2] and the associated white paper[3].

Re-test notes: The revised contracts after the initial report was delivered are in commit `ccf03148d107634b02ad7969db3bb954825f8ea2`.

The audit identified a variety of issues with impacts ranging from very minor to moderate. The most significant finding addresses concerns that the contract owner can arbitrarily change the pool of reserve tokens for the year+2 sale by repeatedly calling a specific function.

A majority of the code was standard and copied from widely-used and reviewed contracts and, as a result, many of the findings pertained to the custom portions of code related to the contract defining the behavior of the crowdsale itself.

Re-test notes: the moderate issue uncovered by the initial review of the contracts was completely fixed by the Rouge Project only leaving very minor findings.

General Discussion

The `RGEToken` and `RougeTGE` contracts implement a standard EIP20 token with symbol `RGE` and a crowdsale (here named `TGE` as Token Generation Event). The `RGE` token is a utility token giving the possibility to use the Rouge Network protocol as described in the white paper. The `RGE` token relies on code copied from https://github.com/ConsenSys/Tokens to implement the `EIP20` token from which it inherits.

The code specifies Solidity version 0.4.18, which is older than the the recommended latest version at the time of the audit, which is 0.4.23. The Rouge Project should use the latest version of Solidity in order to take advantage of the latest features and improvements that Solidity has to offer.

New Alchemy discovered that the contracts implemented do not use SafeMath (from conditions that could result in unsafe behavior, without having to consider for each case whether those conditions are possible.

Re-test notes: The Solidity version specified in the audited contracts was updated to version 0.4.23 which is the current latest version as of updating this document. Additionaly, New Alchemy noted that The Rouge Project took this opportunity to also update the `EIP20` code from https://github.com/ConsenSys/Tokens to its latest version. This version `EIP20` provides minor tweaks to make the code compliant with the language updates from Solidity v0.4.21 (events broadcasted through the `emit` keyword).

Contract / Whitepaper / Website Token Coherence

This section describes how accurately the values from the whitepaper are implemented in the actual contracts. Contracts should aim to implement as closely as possible the various descriptions found in the whitepaper and website. For this section, the white paper is going to be considered as the reference for all values. The amount of discrepancies found between white paper and actual contracts should help users decide the level of trust they can put into the contracts as they are implemented.

Item name	White Paper Value	Smart contract Value	Discrepancy (%)
total supply	1,000,000,000	1,000,000,000	**0%**
TGE distribution	500,000,000	500,000,000	**0%**
reserve pool y+1	300,000,000	300,000,000	**0%**
reserve pool y+2	200,000,000	200,000,000	**0%**
price per token*	$0.076 USD = 1 RGE	set at TGE start	**N/A**

*: The white paper and website indicate that the price per RGE token of $0.076 USD will be used to determine the price in ETH at TGE (Token Generation Event). The TGE contract effectively implements a one time set fixed price for RGE tokens determined at contract creation on the blockchain (as the accurate price of a token in ETH can't be predicted before the TGE and all transactions are made in ETH). In this respect the

any transfers to the contract to ensure the price was accurately set. That price can not be changed by anyone (including contract owner) once the contract has been created.

The Rouge Project accurately implemented in the `RougeTGE` contract all the above listed predetermined values from the white paper and website.

Moderate Issues

Fixed: Contract Owner Can Arbitrarily Alter y+2 Reserve Pool

The `endCrowdsale` function of the `RGEToken` contract allows the contract owner to end the crowdsale, transferring the provided `unsold` amount to the `reserveY2` pool. Investors must trust that the crowdsale owner calls this function with the correct `_unsold` value. Additionally, the `endCrowdsale` function can be called by the owner multiple times, allowing them to arbitrarily increase the y+2 reserve pool. Because this operation uses the built-in integer addition operator, the owner can also decrease the reserve y+2 pool by supplying a large enough value to overflow the uint256 type.

`endCrowdsale` should check that the crowdsale has not already ended, by adding a requirement, such as `require(crowdsale != address(0))`.

Re-test notes: The Rouge Project effectively fixed this issue by removing the `_unsold` function parameter and using instead the internal value held in `balances[crowdsale]`. The Rouge Project also added the recommended requirement (`require(crowdsale != address(0))`) to prevent multiple calls to endCrowdsale regardless of sale status.

Minor Issues

Not Fixed: Lack of Safe Arithmetic

The Rouge Project contracts perform critical integer arithmetic using the default library for all critical integer operations represents a solid defense-in-depth strategy and proactively prevents attacks which exploit unsafe arithmetic.

Re-test notes: The Rouge Project opted not to apply this recommendation. New Alchemy did not find any instances of immediately exploitable unsafe arithmetic in the audited contracts.

Not Fixed: Lack of Short-address Attack Protections

Some Ethereum clients may create malformed messages if a user is persuaded to call a method on a contract with an address that is not a full 20 bytes long. In such a "short-address attack", an attacker generates an address whose last byte is 0x00, then sends the first 19 bytes of that address to a victim. When the victim makes a contract method call, it appends the 19-byte address to `msg.data` followed by a value. Since the high-order byte of the value is almost certainly 0x00, reading 20 bytes from the expected location of the address in `msg.data` will result in the correct address. However, the value is then left-shifted by one byte, effectively multiplying it by 256 and potentially causing the victim to transfer a much larger number of tokens than intended. `msg.data` will be one byte shorter than expected, but due to how the EVM works, reads past its end will just return 0x00.

This attack effects methods that transfer tokens to destination addresses, where the method parameters include a destination address followed immediately by a value. In The Rouge Project contracts, such methods include:

```
- EIP20.transfer
- EIP20.transferFrom
- EIP20.approve
```

While the root cause of this flaw is buggy serializers and how the EVM works, it can be easily mitigated in contracts. When called externally, an affected method should verify that `msg.data.length` is *at least* the minimum length of the method's expected arguments (for instance, `msg.data.length` for an external call to `transfer` should be at least 68: 4 for the hash, 32 for the address (including 12 bytes of padding), and 32 for

- Compare the first four bytes of `msg.data` against the method hash. If they don't match, then the call is internal and no short-address check is necessary.
- Avoid creating `public` methods that may be subject to short-address attacks; instead create only `external` methods that check for short addresses as described above. `public` methods can be simulated by having the external methods call `private` or `internal` methods that perform the actual operations and that do not check for short-address attacks.

Whether or not it is appropriate for contracts to mitigate the short-address attack is a contentious issue among smart-contract developers. Many, including those behind the OpenZeppelin project, have explicitly chosen not to do so. While it is New Alchemy's position that there is value in protecting users by incorporating low-cost mitigations into likely target functions, The Rouge Project would not stand out from the community if they also choose not to do so.

Re-test notes: The Rouge Project opted not to apply this recommendation.

Line by line comments

This section lists comments on design decisions and code quality made by New Alchemy during the review. They are not known to represent security flaws.

RGEToken.sol

Fixed: Lines 24–26: literal value instead of decimals

The values used here represent the number of decimals of the token as defined a few lines above by the `decimals` variable (line 17). To avoid errors if the code needs to be modified and for better readability, all literal values representing the number of decimals of the token should use the defined variable.

Re-test notes: The Rouge Project replaced all instances of literal values representing `decimals` by the variable name.

While none of these operations can be exploited to provoke an integer overflow or underflow under regular operating conditions, using SafeMath would clearly show these are being safeguarded against such issues.

Re-test notes: The Rouge Project opted not to apply this recommendation.

Fixed: Lines 39, 47, 57: startCrowdsaleYx functions can be called with a crowdsale parameter equal to address(0)

The `crowdsale` address variable is used throughout the `RGEToken.sol` contract to hold current tokens to be sold through the various sale steps and to check if the sale is currently ongoing or has been ended though the `endCrowdsale` function. the `crowdsale` variable can be considered to be in two distinct possible states:

- when it is set to `address(0)` the sale is in a non-started state, either because no sale has been started or because the `endCrowdsale` function has been successfully called.
- when it is set to anything else than `address(0)` the sale is in a started state. This happens when a `startCrowdsaleYX` function is called successfully.

The functions `startCrowdsaleY0` , `startCrowdsaleY1` , and `startCrowdsaleY2` each take the new `_crowdsale` address as a parameter, however no check is made to ensure that the parameter passed is different from `address(0)` . If the function is called with such a paramameter, no successful call can be made to `endCrowdsale` , the only solution is to call the `startCrowdsaleYx` function again to set `crowdsale` to the correct address.

The 3 affected functions should add the following requirement to avoid any issues:

```
require(_crowdsale != address(0));
```

Re-test notes: The Rouge Project fixed this issue by adding the recommended requirement.

The values used here represent the number of decimals of the token. All these values should be replaced by a unique named variable (e.g.: `decimals`) to avoid errors if the code needs to be modified and for better readability.

Re-test notes: The Rouge Project replaced all instances of literal values representing `decimals` by the variable name.

Not Fixed: Lines 130–132, 140, 141, 154–162, 176–179: potentially dangerous math operations not using SafeMath

While none of these operations can be exploited to provoke an integer overflow or underflow under regular operating conditions, using SafeMath would clearly show these are being safeguarded against such issues.

Re-test notes: The Rouge Project opted not to apply this recommendation.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bugfree status. The audit documentation is for discussion purposes only.

New Alchemy is a leading blockchain strategy and technology consulting group specializing in tokenization. One of the only companies to offer a full spectrum of guidance from tactical technical execution to high-level theoretical modeling, New Alchemy provides C-level strategy, smart contract development, project management, token design, marketing services, and security audits to the most innovative startups worldwide. ***Get in touch with us at Hello@NewAlchemy.io***

Sign up for exclusive news and analysis from New Alchemy.