Ledgerium

Security Assessment: Final Report

LedgeriumToken Contract

January 10, 2018

Prepared For:

Faisal Mehmood | Ledgerium

faisal@ledgerium.net

Prepared By:

Paul Vienhage | Authio

paulv@authio.org

Contents

Summary

The Ledgerium project has successfully produced a secure token, with no unexpected security problems in the code. Two security problems have been identified with the standards and protocols of the token. (1) The standard ERC20 race condition problems and (2) a single private key controlling the whole of the token supply. Ledgerium's token contains a mitigation of the first problem, and Authio recommends that all users use `decreaseApproval` and `increaseApproval` instead of approve. Ledgerium has provided a plan to secure the tokens they control which I have reviewed. If followed it will help mitigate the risk malicious parties gaining token access and causing loss of token value.

Overall the Ledgerium team has produced a secure and efficient code, and once the small number of findings and recommendations in this report have been addressed the Ledgerium contracts present no immediate reason to delay use.

Scope

Ledgerium provided the following token code for review: https://github.com/ledgerium/pubdocs/blob/master/LedgeriumToken.sol The audit began at commit `53a32dd52017d55171949d71058473af3b1206f7`.

The scope of the audit is limited to that solidity file.

The code (https://github.com/ledgerium/pubdocs/blob/e031c4c94621ef16316ebf682a64c06bc0d1fcc6/LedgeriumToken.sol) was produced in response to the findings and recommendations in the audit.

System Overview

The Ledgerium token contract maintains the following records:

- Balances, a mapping which stores for each address the amount of tokens that they own.
- Allowances, a mapping to a mapping which stores the amount of tokens any address is allowed to move on behalf of another address.
- A set of metadata about the token, including the symbol, the name, and the number of decimals for the token.
- The total supply which is the total outstanding amount of tokens which goes up on minting and down on burning of token.
- A public constant integer which contains the amount of token Ledgerium initially minted.

Token transfer dynamics overview:

The contract allows for two methods of moving tokens. The first is a direct transfer where a user specifies an address and the amount of token to give the address. If the sender has enough token the amount specified is removed from the balance of the sender and added to the balance of the person, they specified.

The second is an indirect transfer where another party moves token on behalf of the token's owner. To use this system a person submits a transfer from request with the owner, the person who should receive the token, and the amount of token to transfer. If the sender is approved to move that much token on behalf of the the owner and the owner has enough token, then the amount of token is removed from the owners balance and added to the receivers balance.

Approval dynamics overview:

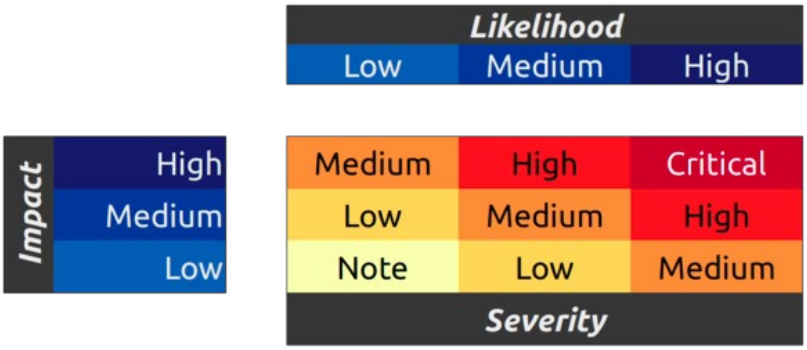
The ERC20 standard requires that an approve method be exposed which allows a user to directly set the allowance for other address. This method takes an address and amount to set the approval to. Unfortunately, it contains a well-known race condition vulnerability allowing extra tokens to be spent by an approved address when the set approval method is called on nonzero values.

For this reason, the Ledgerium contract also contains a pair of methods `increaseApproval` and `decreaseApproval` which increase an address approval by an amount or reduce and addresses approval by an amount when called. These methods mitigate the worst consequences of the race condition and should always be used instead of the approval setting method.

Token creation and destruction:
Tokens in the Ledgerium contract are only created when the contract is created and the tokens given to a single address.
Tokens in the Ledgerium system can be “burned” or destroyed which removes them from the balance and emits a transfer to zero event. There are two ways to do this, the first is that the token owner can specify the amount of tokens they would like to burn and if they have enough token those tokens are transferred to zero. The second is that an approved spender can burn tokens on behalf of the owner.

Findings

The findings section of this report is for explicit security problems with quantifiable possible effects. For this reason each finding is ranked using a variant of the OWASP Risk Rating which rates each vulnerability based on how much damage it can do versus how likely it is that the damage will occur. The following chart details the classification:



ERC20 Race Condition **(Medium Severity):**
This bug is a well know result of implementing the ERC20 standard. Since Ethereum transactions take time to process and the order they are included in is determined by the miners based on gas fees paid, advertisers can call methods knowing that functions will be called after their call. In this case if an addresses approval is set to N and the owner calls the “approve” method to reset it to M, then the approved address can see this and call the `transferFrom` function to move N tokens before the owner sets their approval to M. This allows them to move N+M tokens, which was likely unintended by the approver.
The Ledgerium contract offers alternative methods `increaseApproval` and `decreaseApproval` which are intended to mitigate this problem. However, the method approve still exists in the contract to accommodate the standard so this problem must still be noted.

The `decreaseApproval` methods prevents the most extreme example of the potential abuse for the race condition, though its consequences are still relatively minor. If an

address which has approval N see that the owner has called the method to reduce their approval by M they can race this transaction and spend all N of the token. This has a high likelihood of happening and a low impact [since the approved address must have been approved at some time for N token to get N].

Single Key Balance Holding **(Medium Severity):**
The deployment code for the project mints the total supply of the token to a single address, which is the sender of the deployment code. Despite apparent security in the cryptographic sense, this is not secure from a holistic view of security. There have been a number of incidents where malicious parties have gained access to keys. Such as kidnaping for crypto, focused hacking of developers and executives of companies with ICOs, and even situations like the PRL token hack via corrupt developers. A situation where the ICO's total balance is held on a single or even small group of private keys has a low chance of a devastating attack on token value.

Remediation for this problem is for the Legerium organization to define and implement a plan which holds the majority of funds in a secure way. It should include elements like: use of hot and cold storage, physical keys with strong physical protection, threshold or multi-sig holding of accounts so that funds are released upon consensus of a group.

Mitigation: The Ledgerium team has produced a document which details their internal procedure for key management. If followed it will help to mitigate risk of key theft as it includes multi-sigs, hard wallets, and storage in physically secured locations. I have made some internal suggestions to the Ledgerium team, which if they follow, they will reduce the risk of key theft further. It is not possible to completely mitigate this threat, but Ledgerium has taken steps in the right direction and built a satisfactory security procedure.

Recommendations

The recommendations section is for code recommendations which don't have direct impacts on security but that will improve gas efficiency, make the code more readable, improve adherence to best standards, and otherwise improve the codebase. They contain a rating called priority which is a weighing of the impact of the change vs how much work it is to change the code. A low priority issue has a one to one ratio of payoff to work; note has less; and medium, high, and urgent are all a greater than a 1 to 1 ratio.

Use public and constant data types and remove getter functions **(Priority: Medium)**
The code uses private data types but then exposes getter functions for those same variables. You should use just public data types because they automatically add a getter for you. The pattern of private with a public getter makes the code less readable. This applies to balances, approvals, total supply, and metadata.

Moreover 'ERC20Detailed' should be removed and in the 'LedgeriumToken' use public constants for the token symbol, name, and number of decimals. Since these data points shouldn't change you shouldn't put them into storage; making them public constants is more efficient.

Eliminate Inheritance **(Priority: Medium)**
The contract inheritance pattern is good for development and extension of tokens, but it is not optimal for deployment. Because of the L3 linearization performed by solidity when handling inheritance can introduce weird and hard to find bugs, it is best practice to eliminate inheritance when it is not strictly needed. For example, consider this underhanded solidity entry and (post)[https://pdaian.com/blog/solidity-anti-patterns-fun-with-inheritance-dag-abuse/]. Elimination of inheritance can also reduce the amount of code on chain and thus cost less depending on solidity version and other factors.

Mitigation: Inheritance has been largely eliminated from the contracts, and multiple inheritance was eliminated. This is a satisfactory remediation of this issue.

Make all of the ERC20 function external **(Priority: Low)**
The ERC20 interface specifies external functions so to match it the functions in the ERC20 contract should be made external. Additionally, external functions use slightly less gas because they do not need to support being publicly accessible.

Update to 5.2 compiler version **(Priority: High)**
The modern compiler versions have better security, lower gas usage, and projects produced with versions 0.5.0+ will have more difficulty interfacing with the version 0.4.26 project. For these reasons we recommend upgrading whenever possible.

Mitigation: The compiler has been upgraded to version 0.5.1 which is a satisfactory mitigation of this issue.