My Wish

ERC20 Token Audit

October 2018

https://mywish.io

Audit by
authio

Auditor: Alexander Wade

Contents

1. Overview

MyWish asked us to perform a review of their ERC20 Token contract code. We performed a review of their code from October 5 to October 19, and publish this document as a write-up of our findings.

2. Introduction

2.1 Authenticity

The contracts audited are a subset of the contracts located in the public ico-crowdsale repository. The assessed MyWish smart contract components were written in Solidity, and the version used for this report is commit: **02ba49725e75037dd19182608babcdc92d3e4d0e**.

2.2 Scope

The audit reviewed contracts generated from the repository by an included preprocessor script. Parameters for the preprocessor script were provided and used as directed by MyWish to generate the final audit target: a flattened Solidity file containing several imported smart contracts. The review performed did not assess the preprocessor script, nor did it review any contracts individually. All contracts were reviewed in the context of the flattened file, which included two of the contracts in the github repository, as well as all their inherited components.

The review performed did not assess any scripts, tests, or other non-Solidity files located in the repository. The code reviewed was commented thoroughly during the audit process and is linked in the *Documents and Resources* section.

2.3 Methodology

This audit was performed as a comprehensive review of the codebase and takes into consideration both the Solidity code, as well as the target platform: the Ethereum main network. The Solidity was reviewed not just for common vulnerabilities and anti-patterns, but also for its parity with the intent of the deployer, for its efficiency, and for the practices used during development.

2.4 Risk Assessment

Findings were categorized using a risk rating model based on the OWASP method. Each vulnerability takes into consideration the impact and likelihood of exploitation, as well as the relative ease with which the vulnerability is resolved; findings that permeate throughout the codebase will require much more review and work to solve and are rated higher as a result.

2.5 Disclaimer

This document reflects the understanding of security flaws and vulnerabilities as they are known to Authio, and as they relate to the reviewed project. This document makes no statements on the viability of the project or the safety of its code. This audit does not represent investment advice and should not be interpreted as such.

3. Findings

3.1 Critical Severity

No critical-severity vulnerabilities were found.

3.2 High Severity

No high-severity vulnerabilities were found.

3.3 Medium Severity

No medium-severity vulnerabilities were found.

3.4 Low Severity

Disparity of expectation in release functions: Users use `releaseOnce()` and `releaseAll()` to release their frozen tokens once the freeze period has elapsed. In the event a user does not hold any frozen tokens eligible for release, the `releaseOnce()` function reverts state changes. This is not the case for `releaseAll()`, which will simply do nothing. While this does not pose a significant danger for users, we recommend the inconsistency be addressed.

Overuse of public function visibility: The reviewed token contract is assembled using a script which generates a file of constants with which the token contract will set its initial values. Because each constant is marked `public`, Solidity implicitly creates a publicly visible getter function with the same name. While using constants is generally efficient, excessive use of `public` fields:

1. Makes a contract more expensive to deploy (longer bytecode)
2. Makes a contract more expensive to use, as each additional function selector created by these implicit getters means more options to traverse at runtime.

Consider removing the word `public` from each constant unless absolutely necessary. They will be set to the default, `internal`, meaning they will still be accessible internally to the contract.

3.5 Notes and Recommendations

Important - improper input sanitization during key generation, and mixing of user frozen token records:
The reviewed token contract inherits from `FreezableToken.sol`, an extension of the ERC20 standard implementing token transfers that can transfer time-locked tokens. Users are able to use the function `freezeTo(address, uint, uint64)` to transfer tokens to an address which cannot be transferred again until the specified time period has elapsed.

Frozen tokens are saved in order of release date, for more efficient access; a user must invoke `releaseOnce()` or `releaseAll()` to release their tokens once the period has elapsed. Release dates are linked together through the generation of a unique key using an internal key generator, `toKey(address, uint)`. As the input parameters suggest,

the key is generated using the frozen token holder's address, and the release date. The purpose of this is to ensure that if frozen tokens are sent to a user multiple times with the same release date, the keys generated will be the same each time.

Each generated key is linked to the next sequential release date of a user's frozen tokens in the `chains` mapping. The actual amount frozen for each release date is located in the `freezings` mapping. Both take as input the generated key for a release date. `chains` returns the next date in the sequence, while `freezings` returns the number of tokens frozen at the given release date. To clarify: when a user is sent frozen tokens, a key is generated in `toKey` using their address and the release date. The number of frozen tokens is recorded under this key in `freezings`. If the user has tokens frozen at a later release date than this latest batch, the `chains` mapping will point to the next release date in the sequence, from which a new key can be calculated using `toKey`, which will point to its own amount of frozen tokens and next sequential date if it exists. The pattern repeats until `chains[key]` returns 0, at which point the contract knows it has arrived at the end of the user's frozen token sequence.

The security of each user's frozen token holdings relies on each key generated as being unique. If this were not the case, an attacker could craft an address and release date combination whose generated key matched the key created by a different user to store a record of their frozen tokens, allowing them to lay claim to tokens held frozen by other people.

The key generation used is the crux of the problem: the key generated is 32 bytes in size, the default size used by Solidity for most values. The components of the key used, if the key is to remain unique, should add up to 32 bytes in size. On the surface, this appears to be the case: an initial mask (4 bytes) is followed by the holder's address (20

bytes), followed by the release date, which is assumed to be 8 bytes in size, completing the 32-byte key.

The `toKey` function, however, only *assumes*, but does not require, the provided release date to be 8 bytes in size. Release dates can be given to the function as a default 32-byte value, as `toKey` does not check the size. In the case an attacker is able to provide the function with a 32-byte value, the key generation method used no longer produces unique keys. Instead, the additional bytes overlap with the token holder's address, allowing the attacker to generate release dates that act to completely equate their address with any other address when fed through the key generator. The resulting much later date chosen should impose a waiting period on the attacker, but the release functions only compare the last 8 bytes of the release time with the current time. This means an attacker would be able to access the rightful owner's frozen tokens as soon as they could, for any address and any release date.

Improper input sanitization in `toKey` means that a developer writing or using these contracts needs to remember, every time they use the key generator, to only pass 8 bytes of information into the key generator. In the reviewed code, the input size was correctly altered from its default "32" to the secure "8" a total of 12 out of 12 times. Given that 32-bytes is the default size, this presents a classic anti-pattern - requiring that input sanitization (validating the input size) be performed external to the critical function, rather than simply ensuring the critical function validates the input itself. Compounding this risk is the structure of the `chains` mapping, which only maps from keys to release dates, meaning that multiple users essentially "share" the same mapping. As long as keys are unique, this does not pose a significant risk. However, if an attacker is able to find values for which collisions are created, users sharing the same mapping means the attacker is able to affect the holdings of a much larger proportion of frozen token holders.

Our recommendation is as follows:

1. Do not rely on simple arithmetic operations to generate a unique key. Instead, keys should be generated using `keccak256`, a secure hashing function.
2. Change the `chains` mapping to use separate address spaces for each user. Instead of a mapping from `bytes32 => uint64`, the mapping should map from `address => bytes32 => uint64`. A similar addition should be made to the `freezings` mapping.
3. If the codebase relies on a key generator function like `toKey`, it should check that the input parameters to key generation match the sizes it expects for safe key generation. For example, if dates are only 8 bytes in length the input parameter should read `uint64`, not `uint`; the latter defaults to `uint256`, which uses 32 bytes.

This issue was included as a note, as it does not provide an angle of attack in the reviewed contracts: key generation is successfully handled in the reviewed code. However, we strongly recommend considering and incorporating these changes prior to release and use, and especially if parts of this contract are intended to be used for future deployments.

Redundant modifier use in MintableToken: The modifier `hasMintPermission` is logically equivalent to the `onlyOwner` permission. Consider removing and using `onlyOwner` in all cases.

Unnecessary `boolean` return from public functions: In `MintableToken.mint`, `MintableToken.finishMinting`, and `FreezableMintableToken.mintAndFreeze`, a boolean is returned from the public function. However, logic in these functions dictates that if execution should fail for any reason (insufficient permissions, invalid contract state, etc), then each function will simply revert all state changes. As such, the boolean return value serves to only ever return true; false is never returned. (As an aside, the boolean return values from all ERC20 functions must still be included. While they are just as redundant as their use in this function, the ERC20 standard requires they be included. However, no such standard requires that `mint`, or any other listed functions, returns `true`.

4. Documents and Resources

4.1 Line by line commenting:

Additional notes, suggestions, and comments can be found in the line-by-line comments located here: https://github.com/authio-ethereum/Audits/tree/master/MyWish

4.2 Source:

The original source code used can be found in the MyWish ico-crowdsale repository: https://github.com/MyWishPlatform/ico-crowdsale/tree/v0.7.0

5. Conclusion

Outside of the included notes, the code reviewed was simple and clean. The formatting, naming, and other conventions used were fairly regular, and the inheritance structure was well-organized, resulting in a codebase that was easier to review.