Security Audit of HelloGold Smart Contract

This report is public.

ChainSecurity Ltd.

January 12, 2018

Contents

We first and foremost thank HelloGold for giving us the opportunity to audit your smart contract code. This documents outlines our methodology, limitations and results for your security audit.

1 System Overview

In the following we briefly describe the GoldBackedToken (GOLDX). As the name suggests, the GOLDX represents real gold value on the Ethereum blockchain. GOLDXs are issued whenever an additional allocation is made by the HelloGold Foundation. The allocations happen according to the current share of HelloGoldTokens (HGTs). Consequently, an account with more HGTs receives more GOLDXs.

Furthermore, GOLDXs are subject to a fee. Therefore, their amount reduces over time if no new allocations are made. The fee is calculated before any transactions are made.

In this review, we are auditing GOLDX, which adds some additional features to the original GoldBackedToken and therefore also has to perform a migration from the old to the new token contract.

1.1 Extra Features

Pausable HelloGold has the power to pause and unpause the transfer of tokens.

2 Audit Overview

2.1 Scope of the Audit

The audit was based on the Ethereum Virtual Machine (EVM) after `EIP-150` and solidity compiler `0.4.17+commit.bdeb9e52.Linux.g++`.

The scope of the audit is limited to the following source code files. All of these source code file were received on December 11th, 2017:

- GoldBackedToken2.sol
  - Final SHA-256: `b958dcb3e972cbbeca587c2b897377caff866f9f14dd6ca6385e71d59b65bc85`

2.2 Depth of Audit

The scope of the security audit conducted by ChainSecurity Ltd. was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- 6-hours of manual audit of the contracts listed above for security issues.

2.3 Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorize the findings into 3 distinct categories, depending on their criticality:

- Low - can be considered as less important
- Medium - needs to be considered to be fixed
- High - should be fixed very soon
- Critical - needs to be fixed immediately

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as ✓ No Issue. Finally, if during the course of the audit process, an issue has been addressed technically, we label it as ✓ Fixed, while if it has been addressed otherwise we label it as ✓ Addressed.

Findings that are labelled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

[1] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

3 Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, ChainSecurity Ltd. has performed auditing in order to discover as many vulnerabilities as possible.

4 Details of the Findings

4.1 No Reentrancies ✓ No Issue

The HelloGold contracts do not contain any vulnerabilities that would allow reentrancy attacks. This is because no untrusted code is ever invoked. When function calls are made, they are only directed at trusted code, e.g. fee calculation or old GOLDX.

4.2 No Callstack Bugs ✓ No Issue

The HelloGold contracts do not contain any vulnerabilities that would allow attacks based on a callstack overflow. This is because all exception are properly handled and propagated.

4.3 Ether Transfers ✓ No Issue

The smart contracts do not send or receive ether. There are neither payable functions declared in the contracts, nor calls to `transfer`, `send`, or `call.value()()`. Therefore, there are no issues related to unexpected ether transfers.

4.4 Safe Math ✓ No Issue

HelloGold also uses the popular `SafeMath` library for critical operations to avoid arithmetic over- or underflows and safeguard against unwanted behaviour. In particular, critical variables such as the `amount` within `balances` are only updated using `SafeMath` operations or constants.

4.5 Sum of Tokens smaller than totalSupply ✓ No Issue

Due to arithmetic rounding effects during token allocation and fee calculation the sum of all GOLDX will be smaller than the value returned by `totalSupply`. However, this is not an issue as HelloGold is using `totalSupply` to ensure an upper bound which still holds.

5 Recommendations

- Due to the possible race condition with the `approve` function, the addition of an `increaseApproval` and `decreaseApproval` function has been proposed. These functions eliminate the race condition and do not require two separate transactions to change the approval. Consequently, they could be added to GOLDX.
- Optionally, a check like `require(_to != address(0));` can be used in `transfer` and `transferFrom` to avoid accidental token burning.
- There is a typo in the comment "buring" instead of "burning".
- A function to recover stray ERC20 tokens, which were erroneously sent to the GOLDX contract, could be added.
- The lines:

```
1         balances[source].lastUpdated = now;
2         balances[source].nextAllocationIndex = currentAllocations.length;
```

  inside the `burnTokens` and `mintTokens` functions are redundant, as these operation is also performed during the `update` function, which is previously called inside the respective functions. Removing the lines, therefore saves 40427 gas per function execution.
- The `HelloGoldToken` should be paused during the upgrade to avoid race conditions. This should only be undone after the all setup functions, e.g. `setHGT` have been called.
- Adding `require(partPos == 0);` in `addAllocationPartOne` would prevent any accidental calls to `addAllocationPartOne` by HelloGold which could damage the contract state.
- Obviously, no allocations should be made any longer to the old contract of GOLDX. Otherwise, a double-claim would be possible.

6 Conclusion

The HelloGold smart contracts have been analyzed under different aspects, with different open-source tools as well as our fully fledge proprietary in-house tool. Overall, we found no significant security concerns about the HelloGold smart contracts. Therefore, we believe that the GoldBackedToken can be deployed to the blockchain.

7 Disclaimer