Infinigold Pty Ltd

Smart Contract Security Review Summary

Version: 1.0

July, 2020

Contents

Introduction

Sigma Prime was commercially engaged to perform two time-boxed security reviews of the smart contracts underlying the Infinigold token, a gold-backed stablecoin powered by Ethereum.

This document serves as a summary of the aforementioned reviews.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review regarding, the underlying business model or the individuals involved in the project.

Overview

InfiniGold allows investors to buy, sell and hold physical gold stored at The Perth Mint - a large refining mint - in a digital form. To support this gold-backed stable coin, Inifingold has developed a set of smart contracts on Ethereum which implement the following:

- The ERC20 standard [1];
- Minting functionality, restricted to authorised addresses (i.e. minters);
- Burning functionality, invoked when an ERC20 transfer is made to a specific burn address;
- Proxy pattern to support contract upgrades based on *ZeppelinOS* [2];
- Separate storage contracts for token balance sheet, token allowances and parts of role-based access control;
- Blacklisting feature allowing accounts assigned to the `ListerRole` role to prevent specific addresses from interacting with the smart contracts;
- Whitelisting feature enabling accounts assigned to the `ListerRole` role to allow specific addresses to burn tokens (by transferring to the `burnAddress`).

The related Perth Mint Gold (PMGT) tokens will be redeemable for gold certificates, with operational procedures for exchanging between them. It is understood that the Infinigold stablecoin is *centralised*, since smart contracts include operational roles for facilitating actions such as token minting.

Security Assessment Summary

For specific contract and individual file details, please see the individual reports detailed in the Introduction.

The manual code-review section of the reports, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focuses on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [3, 4].

To support these reviews, the testing team used the following automated testing tools:

- Rattle: `https://github.com/trailofbits/rattle`
- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

The testing team identified a total of eight (8) issues during these two assessments, of which:

- One (1) is classified as critical,
- One (1) is classified as high risk,
- One (1) is classified as low risk,
- Five (5) are classified as informational.

These issues have been addressed and/or acknowledged by the development team and are all considered resolved or closed.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Infinigold smart contract reviews listed in the Introduction. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
INF-01	Inadequate Proxy Implementation Preventing Contract Upgrades	Critical	Resolved
INF-02	Blacklisting Bypass via `transferFrom()` Function	High	Resolved
INF-03	Possible Unintended Token Burning in `transferFrom()` Function	Low	Resolved
INF-04	Slot Location Hash-salt Set to Default	Informational	Resolved
INF-05	Miscellaneous General Comments	Informational	Resolved
INF-06	Denial of Service Vector from Unbound List	Informational	Closed
INF-07	Miscellaneous General Statements	Informational	Resolved
INF-08	ERC20 Implementation Vulnerable to Front-Running	Informational	Closed

INF-01	Inadequate Proxy Implementation Preventing Contract Upgrades		
Asset	TokenImpl.sol		
Status	**Resolved:** See Resolution.		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `TokenImpl` smart contract requires `Owner`, `name`, `symbol` and `decimals`, of `TokenImpl` to be set by the `TokenImpl` constructor.

Consider two smart contracts, `contract A` and `contract B`. If `contract A` performs a `delegatecall` on `contract B`, the state/storage variables of `contract B` are not accessible by `contract A`.

Therefore, when `TokenProxy` targets an implementation of `TokenImpl` and interacts with it via a `DELEGATECALL`, it will not be able to access any of the state variables of the `TokenImpl` contract. Instead, the `TokenProxy` will access its local storage, which does not contain the variables set in the constructor of the `TokenImpl` implementation.

When the `TokenProxy` contract is constructed it will only initialize and set two storage slots:

- The proxy admin address (`_setAdmin` internal function)
- The token implementation address (`_setImplementation` private function)

Hence when a proxy call to the implementation is made, variables such as `Owner` will be uninitialised (effectively set to their default value). This is equivalent to the owner being the `0x0` address. Without access to the implementation state variables, the proxy contract is rendered unusable.

As stated by the ZeppelinOS documentation [2]:

if your logic contract relies on its constructor to set up some initial state, this has to be redone after the proxy upgrades to your logic contract. For example, it's common for logic contracts to inherit from Zeppelin's Ownable contract implementation. When your logic contract inherits from Ownable, it also inherits Ownable's constructor that sets who the owner is upon contract creation. When you link the proxy contract to use your logic contract, the value of who the owner is from the proxy's perspective is lost.

Further information can be found in the Solidity documentation which describes how state variables are laid out in storage as well as this blog post [5] which discusses the `DELEGATECALL` opcode dynamics.

Recommendations

There are number of strategies one can use to mitigate this issue, we provide some examples below:

- Set fixed constant parameters as Solidity constants. The solidity compiler replaces all occurrences of a constant in the code and thus does not reserve state for them. Thus if the correct getters exist for the ERC20 interface, the proxy contract doesn't need to initialise anything.
- Create a constructor-like function that can only be called once within `TokenImpl`. This can be used set the state variables as is currently done in the constructor, however if called by the proxy after deployment, the proxy will set it's state variables.
- Create getter and setter functions that can only be called by the `owner`. Note that this strategy allows the `owner` to change various parameters of the contract after deployment.
- Predetermine the slots used by the required variables and set them in the constructor of the proxy. The storage slots used by a contract are deterministic and can be computed. Hence the variables `Owner`, `name`, `symbol` and `decimals` can be set directly by their slot in the proxy constructor.

Resolution

The development resolved this vulnerability by:

- Updating both `TokenImpl` and `AddressList` to use an `initialize` function instead of a constructor;
- Removing the constructor from the `OwnerRole` contract in favour of calling `super._addOwner()` in the initialize function of child contracts;
- Removed the `BalanceSheet` and `AllowanceSheet` contracts in favour of using the standard `ERC20.sol` OpenZeppelin contract and storing balances/allowances in the proxy (using the *unstructured storage* pattern [2]).

INF-02	Blacklisting Bypass via `transferFrom()` Function		
Asset	TokenImpl.sol		
Status	**Resolved:** See Resolution.		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `transferFrom()` function in the `TokenImpl` contract does not verify that the sender (*i.e. the `from` address*) is not blacklisted. As such, it is possible for a user to allow an account to spend a certain allowance regardless of their blacklisting status.

This vulnerability can be exploited as follows:

1. **Step 1:** ***Mallory sets an allowance of 100 PMG to Bob***
   — `approve(Bob, 100), from{Mallory}`
2. **Step 2:** ***Mallory is blacklisted and should no longer be able to use her balance***
   — `blacklist.addAddress(Mallory)`
3. **Step 3:** ***A `transferFrom()` by Bob can still use Mallory's allowance***
   — `transferFrom(Mallory, Bob, 100), from{Bob}`

This scenario can be exploited by any user with an allowance from Mallory at the time of blacklisting. The user will be able to use the allowance from Mallory as long as there is a balance in Mallory's account.

Furthermore, this vulnerability can be leveraged to completely bypass the blacklisting mechanism. Indeed, a user who controls 2 accounts `account A` and `account B` can allow `account B` to use `account A`'s funds. After performing prohibited/illegal activities, and effectively getting his/her `account A` blacklisted, the malicious user can simply use `account B` to move the funds.

Recommendations

At present the function `transferFrom()` uses the `notBlacklisted(address)` modifier twice, on the `msg.sender` and `to` addresses. The `notBlacklisted(address)` modifier should be used a third time against the `from` address.

Resolution

The development team implemented the recommendation above.

INF-03	Possible Unintended Token Burning in `transferFrom()` Function		
Asset	TokenImpl.sol		
Status	**Resolved:** See Resolution.		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Infinigold allows users to convert/exchange their PMGT tokens to "gold certificates", which are digital artefacts effectively redeemable for actual gold. To do so, users are supposed to send their PMGT tokens to a specific *burn address*.

The `transferFrom()` function does not check the `to` address against this *burn address*. Users may send tokens to the burn address, using the `transferFrom()` function, without triggering the emission of the `Burn(address indexed burner, uint256 value)` event, which dictates how the gold certificates are created and distributed.

The scenario is as follows:

1. **Step 1:** ***Alice set allowance of Bob to 100 PMGT*** — `approve(Bob, 100), from{Alice}`
2. **Step 2:** ***Bob burns token from Alice*** — `transferFrom(Alice, burnAddress, 50), from{Bob}`

Recommendations

We suggest few possible solutions to the unintended burning of tokens in `transferFrom()`.

1. Implement the burning logic in the ERC20 `transfer()` function, forcing all methods to check for the burn address. Equivalently, implement the ERC20 `transferFrom()` logic into `TokenImpl` `transferFrom()` and have it call the `TokenImpl` `transfer()` rather than the ERC20 `transfer()` function.
2. Prevent sending tokens to the burn address in the `transferFrom()` function. This can be achieved by adding a `require` within `transferFrom()` which disallows the `to` address to be the `burnAddress`.
3. Allow and support burning within `transferFrom()`. This can be achieved by adding a logical condition (i.e. an `if` statement) which will either:
   - Call `ERC20.burnFrom()` if the `to` address matches the burn address.
   - Call `ERC20.transferFrom()` for all other `to` addresses.

Resolution

The development team implemented recommendation (2).

INF-04	Slot Location Hash-salt Set to Default
Asset	UpgradeabilityProxy.sol and AdminUpgradeabilityProxy.sol
Status	**Resolved:** See Resolution.
Rating	Informational

Description

Both the `UpgradeabilityProxy` and `AdminUpgradeAbilityProxy` use fixed default keccak hashes to specify the storage slots for the proxy. In particular, the storage slots for each contract are the keccak hashes of `org.zeppelinos.proxy.implementation` and `org.zeppelinos.proxy.admin` respectively.

If a 256-bit hash collision were found for either of these fixed values, it is possible that a user could overwrite the storage slots and take ownership of the proxy. To-date it is considered extremely unlikely that such a collision can be found, however all contracts implementing the default hash would be effected. In this fashion the hash choice for the storage slot behaves like a salt, which should be unique to each contract.

Recommendations

Despite the low probability of a collision being found, we recommend using a non-default hash such that these contracts remain unaffected in case a keccak hash collision of the default OpenZeppelin values is ever discovered.

Resolution

The development team implemented the recommendation above and changed the storage slot hashes to `keccak256("infinigold.proxy.implementation")` and `keccak256("infinigold.proxy.admin")`.

INF-05	Miscellaneous General Comments
Asset	TokenImpl.sol, ERC20.sol, Blacklistable.sol, Whitelistable.sol
Status	**Resolved:** See Resolution.
Rating	Informational

Description

This section describes general observations made by the testing team during this assessment.

1. **Potential Gas Optimisation:**
   - Multiple functions can be declared as `external` to save gas. The comprehensive list of functions that can be updated to have an `external` visibility can be found in the **??** appendix.
   - The following `require` statements in `ERC20.sol` can be removed:
     - line [163], `require(owner != address(0))`
     - line [195], `require(account != address(0))`
   - The `_mint()` and `_burn()` internal functions can be replaced with inline code in `TokenImpl.sol` to save a `JUMP` instruction and a few other opcodes.
   - The `_willFallbackFunction` function can be removed. This would however result in significant changes to the proxy contract.
2. **Unused Modifiers:** The following modifiers are defined but never used:
   - `onlyWhitelisted`;
   - `onlyBlacklisted`;
   - `notWhitelisted`.
3. **Unused Function:** The following `internal` function is defined but never called:
   - `_burnFrom(address account, uint256 value)`.

Recommendations

Consider implementing the following recommendations:

1. Change the visibility of the functions listed in **??** from `public` to `external` to optimise gas costs.
2. Remove the unnecessary `require` statements.
3. Replace the internal functions mentioned above with inline code in the `TokenImpl` contract.
4. Change the proxy contract to be able to remove the `_willFallbackFunction` function.
5. Remove the unused modifiers for improved code readability.
6. Remove or call (see INF-03) the `_burnFrom` function.

Resolution

The development team implemented recommendations (1), (2), (3), (5), and (6).

INF-06	Denial of Service Vector from Unbound List
Asset	Roles.sol
Status	**Closed:** See Resolution.
Rating	Informational

Description

The `reset()` internal function (called by the `replaceAll()` function) resets the `role` linked list by deleting all the elements (*i.e. nodes*) part of the `bearer` mapping.

The caller is bound by the number of elements that are being removed for a particular `role`. Calling the `reset()` function will exceed the current block gas limit (*i.e. 8,000,0000*) for more than 371 total elements in a `role` *linked list*.

Similarly, the `size()` and `toArray()` functions also loop through the linked list to respectively return:

- The number of accounts included;
- An array containing those accounts.

This essentially means that listers, unlisters, minters, pausers, unpausers and owners can perform denial of service attacks on the lists they administer.

In a scenario where the `Roles` library is leveraged by other smart contracts, calling these two functions will also result in a potential denial of service after a certain number of elements have been included in the linked list (this number would depend on the gas cost of the Opcodes implemented by the calling functions).

Recommendations

This issue is raised to make sure the development team is aware of this limitation.

One way to ensure that the current block gas limit is not exceeded would be to introduce a condition in the `add()` function to check that the linked list size is strictly lesser than 371 elements before adding a new element.

This additional condition would significantly increase the gas cost associated with calling the `add()` function, as a call to the `size()` function would be required to fetch the exact number of nodes in the linked list.

Alternatively, the `gasleft()` Solidity special function could be used to make sure that going through the linked list does not exceed the block gas limit.

Finally, the `reset()` could be changed to allow for removing an arbitrary number of nodes (by taking this number as a function parameter).

Resolution

The development team acknowledges this issue and accepts it as a limitation. See below response:

The accounts with access to the various roles will be administered by InfiniGold operators and our procedures will ensure that the number of accounts with access to any given role will be signifincantly less than this upper limit.

INF-07	Miscellaneous General Statements
Asset	Roles.sol
Status	**Resolved:** See inline comments
Rating	Informational

Description

This section describes general observations made by the testing team during this assessment.

1. Both the `remove()` function and `add()` function in the `Roles` library contain redundant `require` statements:
   - Function `add()`:
     - line [74] `require(account != address(0))`.
     - Function `has()`: line [180] `require(account != address(0))` (called by `add()`).
   - Function `remove`:
     - line [94] `require(account != address(0))`.
     - Function `has()`: line [180] `require(account != address(0))` (called by `remove()`).

   ✓ *Resolved: line [74] and line [94] have been removed.*
2. The `CEILING` constant, used to determine the size of the memory array `removedAccounts` in the `Roles` library, can be replaced by a call to the `size()` function to allocate the exact size needed.
   ✓ *Resolved in latest version of the smart contracts (raised directly with the development team during the review)*
3. The variable name `next` can be confusing, as this `address` variable actually references the *previous* linked node.
   ✓ *Resolved in latest version of the smart contracts (raised directly with the development team during the review)*

Recommendations

1. Consider removing redundant requires:
   - Remove line [74] from `add()`;
   - Remove line [94] from `remove()`.

   ✓ *Recommendation implemented*
2. Consider replacing the `CEILING` constant with a call to the `size(role)` function;
   ✓ *Recommendation implemented*
3. Consider changing the variable name `next` to `previous`.
   ✓ *Recommendation implemented*

Resolution

All the recommendations above have been implemented.

INF-08	ERC20 Implementation Vulnerable to Front-Running
Asset	ERC20.sol
Status	**Closed:** See Resolution.
Rating	Informational

Description

Front-running attacks [6, 7] involve users watching the blockchain for particular transactions and, upon observing such a transaction, submitting their own transactions with a greater gas price. This incentivises miners to prioritise the later transaction.

The ERC20 implementation is known to be affected by a front-running vulnerability, in its `approve()` function.

Consider the following scenario:

1. Alice approves Malory to spend 1000 PMGT, by calling the `approve()` function.
2. Alice wants to reduce the allowance, from 1000 PMGT to 100 PMGT, and sends a new transaction to call the `approve()` function, this time with `100` as a second parameter.
3. Malory watches the blockchain transaction pool and notices that Alice wants to decrease the allowance. Malory proceeds with sending a transaction to spend the 1000 PMGT, with a higher gas price, which is then mined before Alice's second transaction.
4. Alice's second `approve` happens, effectively providing Malory with an additional allowance of 100 PMGT.
5. Malory spends the additional 100 PMGT. As a result, Malory spent 1100 PMGT, when Alice wanted her to only spend 100 PMGT.

Recommendations

Be aware of the front-running issues in `approve()`, potentially add extended approve functions which are not vulnerable to the front-running vulnerability for future third-party-applications. See the Open-Zeppelin [8] solution for an example. We note that modifying the ERC20 standard to address this issue may lead to backward incompatibilities with external third-party software.

Resolution

The development team acknowledges this issue and added a `@notice` comment to the `approve` function in the `TokenImpl` smart contract, advising users to only use this function to set the spender allowance to zero.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

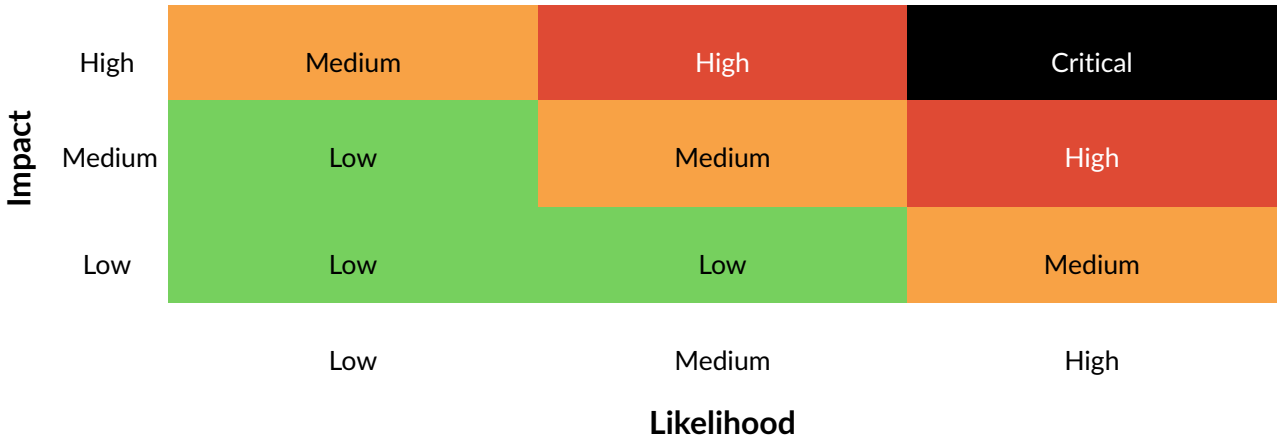

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] ERC-20 Token Standard. Github, Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[2] Open Zeppelin. Proxy Patterns, Available: https://blog.zeppelinos.org/proxy-patterns/. [Accessed 2019].

[3] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[4] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

[5] Sigma Prime. Solidity Security - Delegatecall. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#delegatecall. [Accessed 2018].

[6] Sigma Prime. Solidity Security - Front Running. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#race-conditions. [Accessed 2018].

[7] NCC Group. DASP - Front Running. Website, 2018, Available: http://www.dasp.co/#item-7. [Accessed 2018].

[8] OpenZeppelin StandardToken.sol. Github, 2018, Available: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/v2.1.2/contracts/token/ERC20/ERC20.sol.