PUBLIC

Code Assessment

of the Hoprnet Token Smart Contracts

June 29, 2021

Produced for

hopr

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Sebastian,

First and foremost we would like to thank Hoprnet for giving us the opportunity to assess the current state of their Hoprnet Token system. This document outlines the findings, limitations, and methodology of our assessment.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Sincerely yours,

the ChainSecurity team

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	6
• Code Corrected	5
• Risk Accepted	1

2 Assessment Overview

In this section we briefly describe the overall structure and scope of the engagement including the code commit which is referenced throughout this report.

2.1 Scope

The general scope of the assessment is set out in our engagement letter with Hoprnet dated January 18, 2020. The assessment was performed on the source code files inside the Hoprnet Token repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	Feb	663ed4292cabe218923322133d3058d8cdae86a9	Initial Version
2	Feb	08a82abaf4478d4ec7b42e8a10bdf38cb28d8d8e	Second Version
3	Mar	b89f84e74f314b90d26d615799a27f785f3eba86	Third Version

For the solidity smart contracts, the compiler version `0.6.6` was chosen.

2.1.1 Excluded from scope

All files except for the `HoprToken` and the `HoprDistributor` and their dependencies are out of scope of this audit.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

Hoprnet's token implementation extend the ERC777 with a snapshot ability. Due to the chosen data types at most 3.4 * 10**20 tokens (with 18 decimals) can exist. An additional distribution contract manages different vesting schemes. The Token is mintable by a minter role. The distribution contract calls the mint function to distribute the token and, hence, needs to have the minter role. Additionally, a default admin role exists to grant permissions to the minter role. The token distribution is flexible and one account can be part of different distribution schemes.

All implicit and explicitly defined roles are:

- The user
- The default admin
- The minter
- An operator (ERC 777)
- Allowed spenders (ERC 20)

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of 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 are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of the findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Effects of Snapshotting at Every Block Risk Accepted

5.1 Effects of Snapshotting at Every Block

Design Low Version 1 Risk Accepted

The `HoprToken` performs a state snapshot at every block. That has the following effects:

1. Significant extra gas costs for a token transfer compared to regular token implementations. Even if none of the callbacks are executed, there is an expected overhead of 69,400 gas compared to a regular ERC-20 token and 62,600 compared to a regular ERC-777 token.

Some addresses, e.g. `HoprDistributor` or exchange addresses will amass a considerable number of snapshots. This has two additional effects:

2. The overall contracts state size will be rather big. In case that ETH2.0 transitions to stateless clients, state proofs will be relatively large for all Hopr balances.
3. The gas cost of calling `balanceOfAt` for these contracts with many snapshots will continue to grow. However, as it only grows logarithmically it will foreseeably not reach a critical level. The impact of this is also determined by whether `balanceOfAt` is primarily intended for on-chain or off-chain use.

Risk accepted:

Hoprnet replied:

```
Due to our approach with our upcoming DAO contract, we require a snapshot on every block.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	2

- Burn Function of HoprToken Can Cause Inconsistent Snapshot Code Corrected
- Wrong Check in the HoprDistributor claim Function Code Corrected

Medium-Severity Findings	1

- Miners Can Claim With Schedule Violation Code Corrected

Low-Severity Findings	5

- Multiple Storage Writes Code Corrected
- Redundant Condition Check in _valueAt Code Corrected
- Snapshot Inefficiency Code Corrected
- Superfluous Call to _beforeTokenTransfer Code Corrected
- Timestamp Conversion Has Redundant Operation Code Corrected

6.1 Burn Function of `HoprToken` Can Cause Inconsistent Snapshot

Correctness High Version 1 Code Corrected

The ERC777 has a `_beforeTokenTransfer` hook that is called in the burn, transfer and mint functions. Also it introduced `_callTokensToSend` and `_callTokensReceived` functions that can call the interface implementations registered in ERC1820 registry. ERC777Snapshot utilizes `_beforeTokenTransfer` to track the snapshots after each balance change. Due to the order of `_beforeTokenTransfer` and `_callTokensReceived` functions in the `_burn` function, there is a possibility of reentrancy, that can cause the snapshots to be in an inconsistent state.

```
function _burn(
    address from,
    uint256 amount,
    bytes memory data,
    bytes memory operatorData
)
    internal virtual
{
    require(from != address(0), "ERC777: burn from the zero address");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, from, address(0), amount);
```

```
    _callTokensToSend(operator, from, address(0), amount, data, operatorData);

    // Update state variables
    _balances[from] = _balances[from].sub(amount, "ERC777: burn amount exceeds balance");
    _totalSupply = _totalSupply.sub(amount);

    emit Burned(operator, from, amount, data, operatorData);
    emit Transfer(from, address(0), amount);
}
```

```
function _beforeTokenTransfer(address operator, address from, address to, uint256 amount) internal virtual override {
    super._beforeTokenTransfer(operator, from, to, amount);

    if (from == address(0)) {
        // mint
        updateValueAtNow(accountSnapshots[to], balanceOf(to).add(amount));
        updateValueAtNow(totalSupplySnapshots, totalSupply().add(amount));
    } else if (to == address(0)) {
        // burn
        updateValueAtNow(accountSnapshots[from], balanceOf(from).sub(amount));
        updateValueAtNow(totalSupplySnapshots, totalSupply().sub(amount));
    } else if (from != to) {
        // transfer
        updateValueAtNow(accountSnapshots[from], balanceOf(from).sub(amount));
        updateValueAtNow(accountSnapshots[to], balanceOf(to).add(amount));
    }
}
```

6.1.1 Attack scenario

Attacker can register a IERC777Sender smart contract in the ERC1820 registry that will transfer Hopr Tokens to the attacker. When this transfer happens in the `_callTokensToSend` from ERC1820 registered implementation, the snapshot will be overwritten again, using `balanceOf` value, that has not been yet updated.

```
ERC777Snapshot.burn
└── ERC777.burn
    └── ERC777._burn
        ├── ERC777Snapshot._beforeTokenTransfer
        │   ├── ERC777Snapshot.updateValueAtNow
        │   └── ERC777Snapshot.updateValueAtNow
        ├── ERC777._callTokensToSend
        │   └── Hack.tokensToSend  [CALL]
        │       └── ERC777Snapshot.transfer  [CALL]
        │           ├── ERC777._callTokensToSend
        │           ├── ERC777._move
        │           ├── ERC777Snapshot._beforeTokenTransfer
        │           │   ├── ERC777Snapshot.updateValueAtNow
        │           │   └── ERC777Snapshot.updateValueAtNow
        │           ├── SafeMath.sub
        │           ├── SafeMath.add
        │           └── ERC777._callTokensReceived
        ├── SafeMath.sub
        └── SafeMath.sub
```

In a trace example above, with green color marked balance updates and with yellow - snapshot updates. Due to dependency of snapshot updates depend on balance updates, the reentrancy issue arise. In the other ERC777 functions, such as the transfer function, where the `_callTokensToSend` goes before any state updates, such a problem does not arise. To avoid the issue, the newly released OpenZeppelin contracts should be used.

Code corrected:

The HoprToken now uses an OpenZeppelin ERC777 implementation that does not have a reentrancy vulnerability in its `burn` function. The snapshot is now updated after the external call.

6.2 Wrong Check in the `HoprDistributor` `claim` Function

Security High Version 1 Code Corrected

The `claim` function of the `HoprDistributor` calls internal `_claim` function that contains following code:

```
uint128 newClaimed = _addUint128(allocation.claimed, claimable);
// Trying to claim more than allocated
assert(claimable <= newClaimed);
```

This assertion can only be violated if the `_addUint128` operation overflows. But check of overflow is already present in the `_addUint128`. In addition, there are no checks for the `newClaimed <= allocation.amount`. The comment above the assertion also describes the intention.

Code corrected:

The assertion was rewritten. The new assertion checks that the value of `newClaimed` does not exceed the total allocated amount.

```
assert(newClaimed <= allocation.amount);
```

6.3 Miners Can Claim With Schedule Violation

Security Medium Version 1 Code Corrected

The `claim` function calls the `_getClaimable` function to determine the amount users can claim depending on the elapsed periods.

```
for (uint256 i = 0; i < schedule.durations.length; i++) {
    uint128 scheduleDeadline = _addUint128(startTime, schedule.durations[i]);

    // schedule deadline not passed, exiting
    if (scheduleDeadline > _currentBlockTimestamp()) break;
    // already claimed during this period, skipping
    if (allocation.lastClaim > scheduleDeadline) continue;

    claimable = _addUint128(claimable, _divUint128(_mulUint128(allocation.amount, schedule.percents[i]), MULTIPLIER));
}
```

At the end of `claim` execution, the `allocation.lastClaim` is reassigned, to disable repetitive claims for the same period of the schedule.

```
allocation.lastClaim = _currentBlockTimestamp();
```

But if multiple claims will be send with `block.timestamp` equal to `scheduleDeadline` of some schedule period, multiple repetitive claims of this blocks will be possible. This will effectively allow the hackers to ignore the schedule. While this operation is hard to time right using regular transaction, miners can craft such transactions.

Combined with nonexistent `allocation.amount <= newClaimed` check in `claim` function, this bug also allows to claim more than the allocated amount.

Code corrected:

The condition in the loop was rewritten. Now the equality case will be skipped and repetitive claims for the same periods of the schedule are not possible.

```
if (allocation.lastClaim >= scheduleDeadline) continue;
```

6.4 Multiple Storage Writes

Design | Low | Version 1 | Code Corrected

The `addAllocations` function repeatedly writes to and reads from the `totalToBeMinted` storage variable. This incurs additional gas costs. Note, however, that the additional gas costs will be significantly lowered by the upcoming EIP-2929.

Code corrected:

A new variable `_totalToBeMinted` was introduced. All repetitive operations are performed on it. Thus, gas is saved.

6.5 Redundant Condition Check in `_valueAt`

Design | Low | Version 1 | Code Corrected

The functions `balanceOfAt` and `totalSupplyAt` of the `HoprToken` have following branching conditions:

```
if (
    (accountSnapshots[_owner].length == 0) ||
    (accountSnapshots[_owner][0].fromBlock > _blockNumber)
) {

if (
    (totalSupplySnapshots.length == 0) ||
    (totalSupplySnapshots[0].fromBlock > _blockNumber)
) {
```

In addition, both of these public functions rely on internal `_valueAt` function. Meanwhile the `_valueAt` has following branching conditions:

```
if (snapshots.length == 0) return 0;

if (_block < snapshots[0].fromBlock) {
```

Those conditions are redundant and will never be triggered.

Code corrected:

The checks are now performed only inside the `_valueAt` function.

6.6 Snapshot Inefficiency

Design | Low | Version 1 | Code Corrected

Hoprnet implemented the following binary search:

```
// Binary search of the value in the array
    uint256 min = 0;
    uint256 max = snapshots.length - 1;
    while (max > min) {
        uint256 mid = (max + min + 1) / 2;
        if (snapshots[mid].fromBlock <= _block) {
            min = mid;
        } else {
            max = mid - 1;
        }
    }
    return snapshots[min].value;
```

In case the `_block` number matches the block number of one of the snapshots, the implementation could be optimized. The equality case:

```
snapshots[mid].fromBlock == _block
```

is not handled explicitly. Given that in this case, the result has already been found, there is no need for further unnecessary iterations.

Code Corrected:

The code was adjusted and now explicitly checks for equality:

```
uint256 midSnapshotFrom = snapshots[mid].fromBlock;
if (midSnapshotFrom == _block) {
    return snapshots[mid].value;
```

Hence, the inefficiency is gone.

6.7 Superfluous Call to `_beforeTokenTransfer`

Design | Low | Version 1 | Code Corrected

When overriding the empty parent function `_beforeTokenTransfer` from the ERC777 template, `super._beforeTokenTransfer` gets called. This call has no effect as the parent is empty. Due to current state of Solidity compiler, this call will create unnecessary operations with no effects. Small amount of gas (+-30) will be wasted.

Code corrected:

The superfluous call to the super class was removed.

6.8 Timestamp Conversion Has Redundant Operation

Design Low Version 1 Code Corrected

Function `_currentBlockTimestamp` has a modulo operation that can be dropped. The default behavior of solidity `uint128(X)` conversion achieves the same result and uses less gas.

```
function _currentBlockTimestamp() internal view returns (uint128) {
    // solhint-disable-next-line
    return uint128(block.timestamp % 2 ** 128);
}
```

Code corrected:

The superfluous modulo operation was removed.

7 Notes

We leverage this section to highlight potential pitfalls which are fairly common when working Distributed Ledger Technologies. As such technologies are still rather novel not all developers might yet be aware of these pitfalls. Hence, the mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

7.1 ERC-20 Approve Race Condition

Note Version 1

The ERC-20 standard has a well-known race condition for the `approve` function if both the new and the old approval are non-zero. Hence, a lot implementations add `increaseApproval` and `decreaseApproval` functions which do not have this issue. The Hopr Token does not have such functions. Hence, it is up to users and using smart contracts to avoid the issue.

7.2 Floating Pragma

Note Version 1

The solc version is fixed in the hardhat configuration to version `0.6.6`. However, the files have a floating pragma.

Furthermore, please note the chosen compiler version `0.6.6` has five known bugs.

7.3 Ownership Cannot Be Atomically Transferred

Note Version 1

The specification says:

> allow admin to transfer or revoke their ownership

There is no classical role transfer function inside the contract. The admin can add a new admin and later revoke itself, but not perform an atomic role transfer.

7.4 Schedules Are Specified Using UTF-8 Strings

Note Version 1

The distribution schedules are addressed using UTF-8 strings. UTF-8 strings have well-known security implications, such as characters that look identical to humans, but have a different byte representation or inverse character order. Hence, calls like `addAllocations` could theoretically be referencing a different schedule than expected.

However, as all of the functions setting up allocations can only be executed by the administrators, there is fairly low risk.

7.5 Theoretical Overflow in Binary Search

Note Version 1

In theory the binary search can overflow. This would affect the following computation:

```
uint mid = (max + min + 1) / 2;
```

This could occur as soon as `snapshots.length` would be larger than `2**255`. As this implies that `2**255` snapshots have been taken, which implies that `2**255` blocks have passed, it is irrelevant in practice.