PUBLIC

Code Assessment

of the Payment Channel Smart Contracts

Sep 29, 2021

Produced for

by

Contents

1 Executive Summary

Dear Sir or Madam,

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

The smart contract code was clear and well documented. The HOPRNet team was professional and responsive. During the audit we found a critical security that was fixed after disclosure. All other issues were acknowledged or the code corrected accordingly. We still want to note, since the smart contract is meant to be used by HOPR nodes, issues related to interactions between systems may arise.

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,

ChainSecurity

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	1
• Code Corrected	1
High-Severity Findings	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	8
• Code Corrected	4
• Acknowledged	4

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 assessment was performed on the `packages/ethereum/contracts/HoprChannels.sol` source code file inside the Payment Channel repository based on the yellow paper and documentation provided by client. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	30 July 2021	06eb3b0898c49ab2eb3f69b9eb4f3e51f82bb637	Initial Version
2	17 September 2021	792803f785cb6939818a3066a0b107652d86f0f0	Second Version

For the solidity smart contracts, the compiler version `0.8.3` was chosen.

2.1.1 Excluded from scope

Excluded from scope are any cryptographic properties and operation regarding the objects passed into the functions like the proof of relay. We treat all these objects as valid and correct objects. We did not check the protocol or incentive structure.

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.

HOPR is building a privacy focused network featuring a build-in incentive model. The reviewed `HoprChannels` contract allows nodes to create a payment channels between each other and allow transfer of HoprTokens between them. The transfers are done via tickets, that have a certain predefined probability to win. Winning tickets cause a transfer of tokens between the channels participants. On a code level the channels are unidirectional, meaning channel A->B not equal to B->A.

To keep the winning probability fair, the channel tickets depend on variables that are unknown in advance. For ticket emitted by A for A->B channel, B has a commitment that is unknown to A. And B does not know the proof of relay in advance and has to transfer the message on a network to know it.

Each unidirectional channel has `channelEpoch`, `ticketEpoch`, and `ticketIndex` associated variables, that the ticket emitter is assumed to take in account, during the generation.

The contract has seven functions to manage the payment channels and related tasks.

2.2.1 Announce

The first and simplest function is to announce the existence of a node via an event.

2.2.2 fundChannelMulti

The function is the initial function to set up a payment channels. The logic transfers tokens into the `HoprChannels` smart contract. Not closed balance can top up the balance of channel and closed once in addition can open themselves. For a given addresses A and B, this function can be used to create two unidirectional channels A->B and B->A. Opening a closed unidirectional channel increases its channelEpoch by one and resets `ticketEpoch` and `ticketIndex` to zero.

2.2.3 redeemTicket

This is the central function to receive the rewards for a ticket from the source's payment channel. The tickets needs to have the right `channelEpoch`, `ticketEpoch`. The `ticketIndex` must be greater than the index of previously redeemed ticket. The old commitment needs to be the hash of the new commitment (hash chain in reverse order). Tickets need to be signed by the issuer (source) and the function enforces the correct ticket signature. In the end, the function checks if the ticket is a win (reward payout), increases the ticket index, sets the new commitment and updates the payment channel balances. If earning channel is closed, the tokens are transferred directly to recipient.

2.2.4 initiateChannelClosure

The function will start to close the channel by setting the status to `PENDING_TO_CLOSE` and the closure time counter. The receiver has now the closure time (around two minutes) to claim open winning tickets.

2.2.5 finalizeChannelClosure

After setting a channel into the pending state and the closure period has passed, the function can be called to close the channel. All remaining funds are transferred back to the channel's source. Two events are emitted. The channel's balance and closure time is set to zero. The channel's status is set to `CLOSED`.

2.2.6 bumpChannel

The function can be called by the destination to set a commitment used similar to a commit reveal scheme to determine if a ticket is winning tickets or not. The `ticketEpoch` is increased by one when this function is called.

2.2.7 tokensReceived

Alternative way to fund a channel. Similar to `fundChannelMulti` but as a hook for the ERC777 HOPR token.

2.2.8 Version 2 changes

Announce function in version 2 of the contract requires message sender to submit a public key for its address. The public key is stored on chain. Opening of the channels is only possible when both channel participants have announced their public keys.

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	4

- Compression of Epochs in Struct Acknowledged
- Gas Optimization Acknowledged
- Node Can Set Ticket Index to Arbitrary High Values Acknowledged
- Redundant Sanity Checks Acknowledged

5.1 Compression of Epochs in Struct

Design Low Version 1 Acknowledged

It is unlikely that the channel struct values for `ticketEpoch` or `channelEpoch` will ever reach the maximum number of uint256. HOPRNet should re-evaluate if these values can be bounded e.g. by uint128 and share a storage slot and thus lower the gas consumption of the contract.

Acknowledged

Gas efficiency issues are considered out of scope.

5.2 Gas Optimization

Design Low Version 1 Acknowledged

The contract includes a struct that stores in storage a mapping with all channels and the respective state for each channel as following:

```
struct Channel {
    uint256 balance;
    bytes32 commitment;
    uint256 ticketEpoch;
    uint256 ticketIndex;
    ChannelStatus status;
    uint256 channelEpoch;
    // the time when the channel can be closed - NB: overloads at year >2105
    uint32 closureTime;
  }
```

The Channel struct includes an attribute status which is of type enum ChannelStatus and has only 4 values, therefore occupies only 8 bits in the storage. Given that there is another attribute uint32 closureTime that occupies another 32 bits, these two attribute status and closureTime should be reordered and placed together to optimize the overall storage used by the contract.

Acknowledged

Gas efficiency issues are considered out of scope.

5.3 Node Can Set Ticket Index to Arbitrary High Values

Design | Low | Version 1 | Acknowledged

The ticket issuing node can set the ticket index at will. If this index is set to a value close or equal to max uint, the tickets would be unusable (not redeemable) quickly and the channel would need to be reset.

Acknowledged

The issue has been discussed and it was decided to check the ticket index off-chain.

5.4 Redundant Sanity Checks

Design | Low | Version 1 | Acknowledged

The function fundChannelMulti checks if the two amounts are greater than zero, which are checked later again in the function _fundChannel.

The modifier validateSourceAndDest is executed twice for each call of _fundChannel from fundChannelMulti if one would fund both channels directions.

Acknowledged

Hopr provided the reasoning why it is done this way and that efficiency issues are out of scope.

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	1

- Reentrancy Can Drain Money Code Corrected

High-Severity Findings	0

Medium-Severity Findings	1

- Inconsistent States and Events Code Corrected

Low-Severity Findings	4

- Channel Transition Model Code Corrected
- Redundant Imports Code Corrected
- Token Transfers Inconsistent Code Corrected
- Variables Could Be Labeled Immutable Code Corrected

6.1 Reentrancy Can Drain Money

Security Critical Version 1 Code Corrected

The HoprChannels smart contract uses the ERC777 HoprToken to settle payments. The ERC777 token allows reentrancies during the transfer via sender and receiver hooks. An attacker can utilize this reentrancy to drain the balance of HoprChannels contract. One of the places where this can happen is in the `finalizeChannelClosure` function.

We describe a more elaborate attack and a straightforward attack.

Attack setup: Alice and Bob cooperate. They have created channels between them, Alice has called `initiateChannelClosure` for her channel with Bob, that holds 100 tokens. Bob has valid and yet unclaimed ticket for 75 tokens. Closure time has passed for the Alice owned channel. Alice has a smart contract registered for ERC777 hook. Bob is smart contracts that is registered in the ERC1820 registry for the ERC777 hooks.

- Alice calls `finalizeChannelClosure` with Bob as destination.
- During the call `token.transfer(Alice, 100);` in this function, Alice contract gets called.
    - Alice contract calls to Bob contract.
        - Bob contract calls the `redeemTicket` with valid unclaimed ticket.
            - Channel (Alice, Bob) is spending and (Bob, Alice) is earning.
            - Balance of (Alice, Bob) is decreased by 75. Since the balance of the channel is still 100, the new value will be 25.
            - Balance of (Bob, Alice) is increased by 75.
            - Function `redeemTicket` returns.
        - The Bob contract execution returns to Alice

    - Alice contract returns `finalizeChannelClosure` call.

- The execution continues after the call `token.transfer(Alice, 100);`
- The balance (25 tokens) of Alice owned channel is nullified with `delete` and the status is set to the CLOSED.

As a result of above described schema, the initial 100 tokens of (Alice, Bob) channel will be payed out to Alice, and in addition Bob will get 75 tokens from his ticket claim. Thus instead of 100 tokens 175 tokens were withdrawn.

The straightforward attack would be to reenter multiple time into the `finalizeChannelClosure` as the state variables are changed after the reentrancy possibility. As a general rule, all state dependent operations should be done before the possible reentrancy. Additionally, reentrancies could be completely blocked if not needed.

Code corrected:

In functions that perform transfers of HOPRToken the transfer operations are moved to the end of the functions.

6.2 Inconsistent States and Events

Design | Medium | Version 1 | Code Corrected

Functions using ERC777 transfers can be reentered (a reentrancy does not necessarily need to happen in the same function but in another relevant function in the system.). Some of these functions have a code after the possible reentrancy point. This might become problematic if the logic relies on state variables like in `finalizeChannelClosure`, `redeemTicket`. Besides the more critical reentrancy issue we mentioned, these function's events might be inconsistent or misleading.

For example, in `redeemTicket` event `ChannelUpdate` is emitted. This event uses `spendingChannel` storage variable. Given the `redeemTicket` is called and the ERC777 hook is used to change any storage variable used in these events, the events can emit inconsistent information. This can be done if during the transfer, the hook calls the `bumpChannel` in between.

The same applies to the other places where logic after the reentrancy possibility relies on state variables. We do not know if the client's or third party software will rely on these events. If so, the severity of the issue would be affected.

Code corrected:

In functions that perform transfers of HOPRToken the transfer operations are moved to the end of the functions.

6.3 Channel Transition Model

Correctness | Low | Version 1 | Code Corrected

According to the channels states transition model, the channel in `Waiting for commitment` state cannot be taken into `Pending To Close`. In the smart contract code, such behavior is allowed in the `initiateChannelClosure` function. In addition, the specification provided by HOPRNet also allows such behavior.

Code corrected:

The transition model for channel states has been updated accordingly.

6.4 Redundant Imports

Design | Low | Version 1 | Code Corrected

The `SafeERC20.sol` library is imported twice in line 10 and 11.

Code corrected

The redundant import is removed.

6.5 Token Transfers Inconsistent

Design | Low | Version 1 | Code Corrected

The `HoprChannels` contract has inconsistent use of `SafeERC20` functions. Since the token is known `HoprToken` contract that cannot be changed after the deployment, the use of `SafeERC20` functions is redundant and introduces the unnecessary gas expenses.

```
token.transfer(msg.sender, channel.balance);
token.safeTransfer(msg.sender, amount);
token.safeTransferFrom(msg.sender, address(this), amount1 + amount2);
```

Code corrected

`transfer` is now used consistently.

6.6 Variables Could Be Labeled Immutable

Design | Low | Version 1 | Code Corrected

The keyword immutable and constant can be used to save gas because the compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed.

`token` and `secsClosure` variable can not be changed and can be set to immutable. `FUND_CHANNEL_MULTI_SIZE` could be set to constant (if calculated beforehand) or else immutable as the value is known when compiling the contract and cannot be changed later.

Code corrected

The variables were labeled immutable.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Automated Security Tools

Note Version 1

While performing the audit we found a simple but severe issue which would have been flagged by basic smart contract security tools. Using linters, static analyzers and other tools could prevent these mistakes and increase the overall code quality.

7.2 Compiler Version

Note Version 1

The used compiler version `0.8.3` is six version behind the current version `0.8.9` (including bug fixes).

7.3 Events Emit Complete Channel Struct

Note Version 1

HOPRNet might evaluate if it is necessary to emit the whole channel struct in events. We are not aware of the needs but if not the whole struct it needed, it would be more efficient to only include the relevant parts in the event.

7.4 Pre- and Post Condition Checks

Note Version 1

If gas efficiency is not a priority, checking pre- and post conditions after important operations might be valuable. Consider for example that the contract balance could be queried before and after a transfer and it could be checked if the balance reduced or increased exactly to the expected amount.