



Catalog

Audit

Presented by:

OtterSec

Nicholas R. Putra

Woosun Song

Robert Chen

contact@osec.io

nicholas@osec.io

procfs@osec.io

r@osec.io



Contents

01 Executive Summary	2
Overview	2
Key Findings	2
02 Scope	3
03 Findings	4
04 General Findings	5
OS-CTL-SUG-00 Inconsistent Implementation Of Expiry Check	6
OS-CTL-SUG-01 Possibility Of Redemption After Expiry	7
 Appendices	
A Vulnerability Rating Scale	8
B Procedure	9

01 | **Executive Summary**

Overview

Catalog Finance engaged OtterSec to perform an assessment of the swapper program. This assessment was conducted between August 7th and August 11th, 2023. For more information on our auditing methodology, see [Appendix B](#).

Key Findings

Over the course of this audit engagement, we produced 2 findings total.

We provided recommendations addressing the semantic distinction present in the expiry value of the Hash Time Lock Contract between the Ethereum and Bitcoin implementations ([OS-CTL-SUG-00](#)). Inconsistent semantic interpretation may result in confusion during future development. Additionally, we have proposed utilizing proper validations for the redeem functionality to prevent users from claiming locked tokens after the contract's expiration ([OS-CTL-SUG-01](#)).

02 | Scope

The source code was delivered to us in a git repository at github.com/catalogfi/swapper. This audit was performed against commit [0be2466](#).

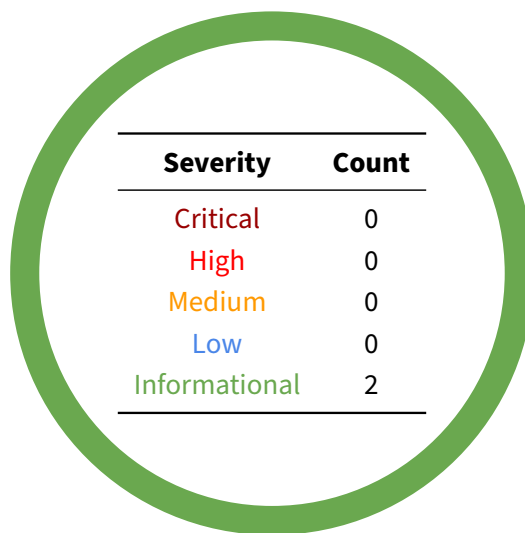
A brief description of the programs is as follows.

Name	Description
swapper	This program implements an HTLC (Hash Time Lock Contract) mechanism for facilitating atomic swaps, enabling signers to generate orders that can be utilized in cross-chain atomic swap transactions.

03 | Findings

Overall, we report 2 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	0
Low	0
Informational	2

04 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-CTL-SUG-00	The meaning of the expiry value of time locks is different for Solidity and Bitcoin scripts.
OS-CTL-SUG-01	Funds that have been locked are still redeemable even after the contract has expired.

OS-CTL-SUG-00 | Inconsistent Implementation Of Expiry Check

Description

There exists a semantic distinction between the Ethereum and Bitcoin implementations concerning the interpretation of `expiry`.

`expiry` in the Ethereum script:

```
contracts/AtomicSwap.sol SOLIDITY

[...]  
require(  
    expiry > block.number,  
    "AtomicSwap: expiry cannot be lower than current block"  
);  
[...]
```

`expiry` in the Bitcoin script:

```
bitcoin/AtomicSwap.ts BTC SCRIPT

OP_ELSE  
    ${bitcoin.script.number.encode(expiry).toString("hex")}  
    OP_CHECKSEQUENCEVERIFY  
    OP_DROP  
    OP_DUP  
    OP_HASH160  
    ${fromBase58Check(initiatorAddress).hash.toString("hex")}  
OP_ENDIF
```

As observed in the Ethereum implementation above, the `expiry` value of the time lock is represented by a specific block number. In contrast, within the Bitcoin script, the `OP_CHECKSEQUENCEVERIFY` opcode is utilized, which compares the time passed until UTXO input creation with the topmost stack element, which in this case is the value of `expiry`. This opcode mandates a time interval to elapse, indicating that the `expiry` is conveyed as a relative time duration.

Remediation

Ensure that both implementations have a consistent semantic interpretation of the `expiry` value to prevent any potential confusion.

Patch

Fixed in [b2c4fe8](#).

OS-CTL-SUG-01 | Possibility Of Redemption After Expiry

Description

The ability to claim locked funds through the redeem functionality persists even after the Hash Time Lock Contract expires, which goes against the intended design of the contract. This concern pertains to both the Ethereum and Bitcoin implementations of the Hash Time Lock Contract.

Remediation

Ensure in both implementations that sufficient validation is executed to ensure locked tokens cannot be redeemed after the contract expires.

Patch

Catalog chose not to remediate this issue to maintain parity with the unalterable Bitcoin implementation.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the [General Findings](#) section.

Critical	<p>Vulnerabilities that immediately lead to loss of user funds with minimal preconditions</p> <p>Examples:</p> <ul style="list-style-type: none">• Misconfigured authority or access control validation• Improperly designed economic incentives leading to loss of funds
High	<p>Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit.</p> <p>Examples:</p> <ul style="list-style-type: none">• Loss of funds requiring specific victim interactions• Exploitation involving high capital requirement with respect to payout
Medium	<p>Vulnerabilities that could lead to denial of service scenarios or degraded usability.</p> <p>Examples:</p> <ul style="list-style-type: none">• Malicious input that causes computational limit exhaustion• Forced exceptions in normal user flow
Low	<p>Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.</p> <p>Examples:</p> <ul style="list-style-type: none">• Oracle manipulation with large capital requirements and multiple transactions
Informational	<p>Best practices to mitigate future security risks. These are classified as general findings.</p> <p>Examples:</p> <ul style="list-style-type: none">• Explicit assertion of critical internal invariants• Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.