hexens × train

FEB.25

SECURITY REVIEW REPORT FOR TRAIN PROTOCOL

CONTENTS

- About Hexens
- Executive summary
    - Scope
- Auditing details
- Severity structure
    - Severity characteristics
    - Issue symbolic codes
- Findings summary
- Weaknesses
    - The LP can steal the funds if the client fails to verify the LP's timelock correctly
    - Discrepancy between the actual locked and stored amount of tokens when using fee-on-transfer tokens
    - Unchecked ERC20 transfer return value may cause HTLC inconsistency across chains
    - Immutable DOMAIN_SEPARATOR becomes invalid after a hard fork
    - Id should be checked by the LP before redeeming
    - ERC20 decimal mismatch may cause incorrect token transfers & fund loss in cross-chain swaps
    - TODO's in code
    - Comment in code could be improved
    - Inconsistent validation of rewardTimelock in lock() function between evm chains and starknet

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit evaluates the Train Protocol (TRustless Atomic INtents), a peer-to-peer system for bridging and swapping crypto assets across blockchain networks. The primary focus is on Atomic Swaps (HTLC - Hashed Time Lock Contract), which facilitate trustless asset exchanges without requiring one blockchain to be aware of the other's state. The process involves the user generating a secret and locking funds on the source chain, while the liquidity provider (LP) locks funds on the destination chain. The secret is then exchanged to unlock the respective funds, with a time limit ensuring fund recovery if the swap fails.

Our security assessment spanned five days and included an in-depth review of three smart contracts—two written in Solidity and one in Cairo.

During the audit, we identified one high-severity vulnerability that could allow the LP to steal user tokens. Additionally, we found two medium-severity issues, two low-severity issues, and four informational findings.

All reported issues were either addressed or acknowledged by the development team and subsequently verified by us.

As a result, we can confidently state that the protocol's security and overall code quality have improved following our audit.

SCOPE

The analyzed resources are located on:

https://github.com/layerswap/layerswap-atomic-bridge/blob/6c96f61d7d6c7e8a8991a12e[...]53b0a9e7b/chains/evm/solidity/contracts/HashedTimeLockERC20.sol

https://github.com/layerswap/layerswap-atomic-bridge/blob/6c96f61d7d6c7e8a8991a12e[...]53b0a9e7b/chains/evm/solidity/contracts/HashedTimeLockEther.sol

https://github.com/layerswap/layerswap-atomic-bridge/blob/6c96f61d7d6c7e8a8991a12e40068ab53b0a9e7b/chains/starknet/src/HashTimeLockedERC20.cairo

The issues described in this report were fixed in the following commits:

https://github.com/TrainProtocol/contracts/commit/f27f0eaf2b2cc784d5746b0ed9cc42ef88241a5a

https://github.com/TrainProtocol/contracts/commit/ee0391e6f864faea0faecf51b5ffbf16d455fb62

AUDITING DETAILS

STARTED
03.02.2025

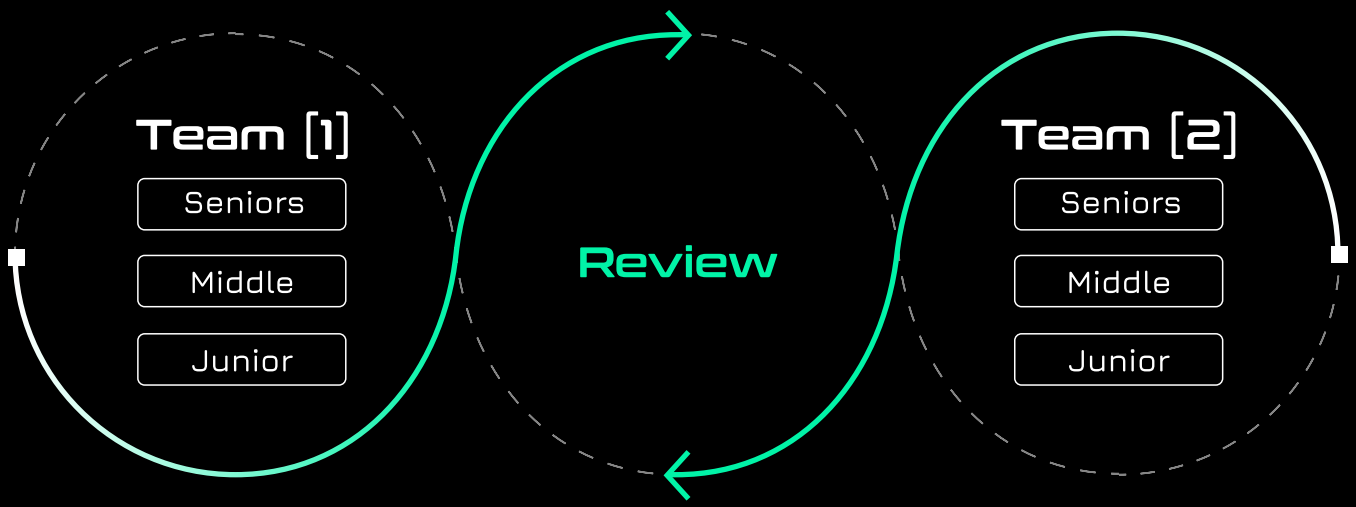
DELIVERED
10.02.2025

Review Led by

TRUNG DINH
Senior Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

ISSUE SYMBOLIC CODES

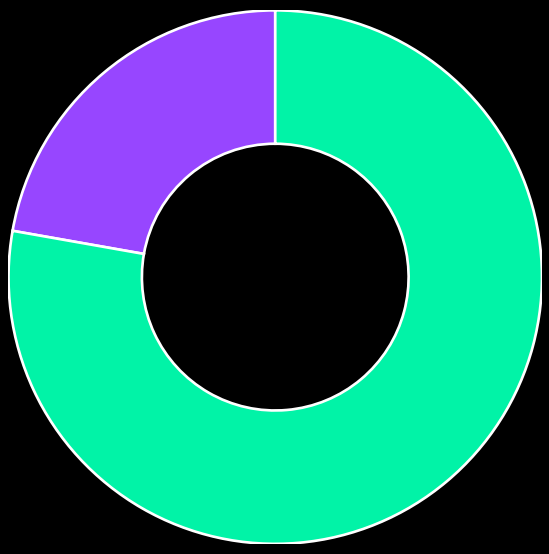
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	2
Low	2
Informational	4

Total: 9

- High
- Medium
- Low
- Informational



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

LYSWP2-7

THE LP CAN STEAL THE FUNDS IF THE CLIENT FAILS TO VERIFY THE LP'S TIMELOCK CORRECTLY

SEVERITY: High

PATH:

chains/evm/solidity/contracts/HashedTimeLockERC20.sol#L172-L176

REMEDIATION:

The issue can be partially mitigated by forcing the LP lock their tokens at least 30 minutes (2 * 15 minutes).

STATUS: Fixed

DESCRIPTION:

The timelock for all user and LP locks is enforced to be at least 15 minutes. However, LPs can exploit this by setting their lock duration to the minimum (15 minutes). This is because users must call **addLock()** after the LPs lock their tokens, ensuring that the user's timelock is always longer than the LP's timelock.

As a result, at the LP's **timelock** expiration, the LP can call **refund()** on the destination chain to reclaim their tokens and then call **redeem()** on the source chain to claim the user's funds. Meanwhile, the user is unable to retrieve their funds since their **timelock** has not yet expired.

Exploit Scenario:

1. Alice (the user) creates a commit object on the source chain, setting Bob (the LP) as **srcReceiver = Bob**.
2. At timestamp T, Bob sees the event on the source chain and locks tokens on the destination chain with a timelock of **T + 900** (15 minutes).
3. Alice must call **addLock()** to finalize her commitment, which always happens after Bob's lock is created. Consequently, Alice's timelock is always longer than Bob's.
4. At T + 900, Bob calls **refund()** on the destination chain to reclaim his funds. At this point, Alice's funds are still locked.
5. Bob then calls **redeem()** on the source chain to claim Alice's funds.

Outcome: Alice loses her tokens, while Bob profits from the exploit.

```
  /// @dev Modifier to ensure the provided timelock is at least 15 minutes
in the future.
  modifier _validTimelock(uint48 timelock) {
    if (block.timestamp + 900 > timelock) revert InvalidTimelock();
    _;
  }
```

LYSWP2-6

DISCREPANCY BETWEEN THE ACTUAL LOCKED AND STORED AMOUNT OF TOKENS WHEN USING FEE-ON-TRANSFER TOKENS

SEVERITY: Medium

PATH:

chains/evm/solidity/contracts/HashedTimeLockERC20.sol#L337-L350

REMEDIATION:

Implement balance checks and store the balanceOf difference in the mapping instead of the user supplied amount.

STATUS: Acknowledged

DESCRIPTION:

To bridge the funds the user needs to lock their funds on one chain to later unlock their funds on another chain. To do so the user can call the **lock()** function in the **HashedTimeLockERC20** contract:

```
if (token.balanceOf(msg.sender) < amount + reward) revert
InsufficientBalance();
if (token.allowance(msg.sender, address(this)) < amount + reward) revert
NoAllowance();

token.safeTransferFrom(msg.sender, address(this), amount + reward);
contracts[Id] = HTLC(
  amount,
  hashlock,
  uint256(1),
```

```
  tokenContract,
  timelock,
  uint8(1),
  payable(msg.sender),
  payable(srcReceiver)
);
```

However as the user given amount is being stored in the **contracts** mapping when using fee-on-transfer tokens the actual amount of tokens that will be transferred and locked in the contract will be less than the value stored in the mapping. This will lead to cases where when the user refunds or actually unlocks their tokens they will use the funds of other users.

If there is low amount of liquidity of that token it can lead to cases where the contract will be unable to refund or unlock the tokens because it will try to transfer the amount stored in the mapping while not having that amount of tokens actually on the contract.

```
if (token.balanceOf(msg.sender) < amount + reward) revert
InsufficientBalance();
if (token.allowance(msg.sender, address(this)) < amount + reward) revert
NoAllowance();

token.safeTransferFrom(msg.sender, address(this), amount + reward);
contracts[Id] = HTLC(
  amount,
  hashlock,
  uint256(1),
  tokenContract,
  timelock,
  uint8(1),
  payable(msg.sender),
  payable(srcReceiver)
);
```

LYSWP2-8

UNCHECKED ERC20 TRANSFER RETURN VALUE MAY CAUSE HTLC INCONSISTENCY ACROSS CHAINS

SEVERITY: Medium

PATH:

chains/starknet/src/HashTimeLockedERC20.cairo#L391-L464, HashTimeLockedERC20.cairo#L339, HashTimeLockedERC20.cairo#L423

REMEDIATION:

Explicitly check all ERC20 transfer return values (transferFrom() and transfer()). If the function returns false, the transaction should revert immediately.

STATUS: Fixed

DESCRIPTION:

The contract does not check the return value of **transferFrom()** and **transfer()** in multiple places, assuming that all ERC20 transfers will succeed. This is an issue because some ERC20 tokens do not revert on failure but instead return false. If the contract does not explicitly validate these return values, it may proceed with logic without actually transferring the funds, leading to severe inconsistencies.

The **commit()** function executes an ERC20 **transferFrom()** call to move tokens from the user to the contract. However, the function does not check the return value of **transferFrom()**, assuming it will always succeed. In cases where the ERC20 token does not revert on failure but instead returns **false**, the function will continue execution without actually transferring funds.

This issue becomes particularly critical in a cross-chain scenario where an LP (Liquidity Provider) observes a commit event and locks funds on the destination chain (e.g., Ethereum). If the initial token transfer on the source chain (e.g., Starknet) failed silently, the LP would lock funds on the destination chain without receiving corresponding funds on the source chain.

This issue is not limited to the **commit()** function. Other functions, such as **lock()**, **redeem()**, and **refund()**, also fail to check ERC20 transfer return values, potentially leading to similar inconsistencies and unexpected behavior.

```
fn commit(
            ref self: ContractState,
            Id: u256,
            amount: u256,
            sender_key: felt252,
            dstChain: felt252,
            dstAsset: felt252,
            dstAddress: ByteArray,
            srcAsset: felt252,
            srcReceiver: ContractAddress,
            timelock: u64,
            tokenContract: ContractAddress,
        ) -> u256 {
            //Check that the ID is unique
            assert!(!self.hasHTLC(Id), "Commitment Already Exists");
            assert!(self.validTimelock(timelock), "Invalid TimeLock");
            assert!(amount != 0, "Funds Can Not Be Zero");

            // transfer the token from the user into the contract
            let token: IERC20Dispatcher = IERC20Dispatcher {
contract_address: tokenContract };
            assert!(token.balance_of(get_caller_address()) >= amount,
"Insufficient Balance");
            assert!(
                token.allowance(get_caller_address(),
get_contract_address()) >= amount,
                "Not Enough Allowence"
            );
            token.transfer_from(get_caller_address(),
get_contract_address(), amount);
```

```
        //Write the PreHTLC data into the storage
        self
            .contracts
            .write(
                Id,
                HTLC {
                    amount: amount,
                    hashlock: 0,
                    secret: 0,
                    tokenContract: tokenContract,
                    timelock: timelock,
                    claimed: 1,
                    sender: get_caller_address(),
                    sender_key: sender_key,
                    srcReceiver: srcReceiver,
                }
            );

        let hop_chains = array!['null'].span();
        let hop_assets = array!['null'].span();
        let hop_addresses = array!['null'].span();

        self
            .emit(
                TokenCommitted {
                    Id: Id,
                    hopChains: hop_chains,
                    hopAssets: hop_assets,
                    hopAddress: hop_addresses,
                    dstChain: dstChain,
                    dstAddress: dstAddress,
                    dstAsset: dstAsset,
                    sender: get_caller_address(),
                    srcReceiver: srcReceiver,
                    srcAsset: srcAsset,
                    amount: amount,
                    timelock: timelock,
                    tokenContract: tokenContract,
                }
            );
        Id
    }
```

LYSWP2-5

IMMUTABLE DOMAIN_SEPARATOR BECOMES INVALID AFTER A HARD FORK

SEVERITY: Low

PATH:

chains/evm/solidity/contracts/HashedTimeLockERC20.sol#L37-L47

chains/evm/solidity/contracts/HashedTimeLockEther.sol#L35-L45

REMEDIATION:

Consider using the implementation from OpenZeppelin, which recalculates the domain separator if the current block.chainid is not the cached chain ID: openzeppelin-contracts/contracts/utils/cryptography/EIP712.sol at 441dc141ac99622de7e535fa75dfc74af939019c · OpenZeppelin/openzeppelin-contracts

STATUS: Fixed

DESCRIPTION:

The **DOMAIN_SEPARATOR** in the **HashedTimeLockERC20**, **HashedTimeLockEther** contracts are defined as a constant and remains unchanged after contract deployment. However, if a hard fork occurs, the **block.chainid** on one of the forked chains will change. This approach is potentially unsafe as such signatures could be deemed valid in forked networks, thus creating a vulnerability to replay attacks.

```
  constructor() {
    DOMAIN_SEPARATOR = hashDomain(
      EIP712Domain({
        name: 'LayerswapV8',
        version: '1',
        chainId: block.chainid,
        verifyingContract: address(this),
        salt:
0x2e4ff7169d640efc0d28f2e302a56f1cf54aff7e127eededda94b3df0946f5c0
      })
    );
  }
```

LYSWP2-13

ID SHOULD BE CHECKED BY THE LP BEFORE REDEEMING

SEVERITY: Low

PATH:

chains/evm/solidity/contracts/HashedTimeLockERC20.sol#L253-L272

REMEDIATION:

Documentation should add the verification of the ID as an important step in the Solver implementation guide.

STATUS: Fixed

DESCRIPTION:

When a user calls the **addLock()** function, an event containing the HTLC ID, hashlock, and timelock is emitted. The Solver implementation documentation recommends verifying the correctness of the hashlock and timelock.

However, if the Solver does not verify which HTLC ID the hashlock and timelock correspond to and only checks their validity, an attacker could deceive the LP by associating the hashlock with a different HTLC—one that locks a lower amount than originally committed. This exploit would enable the attacker to receive significantly more tokens on the destination chain than they locked on the source chain.

```
function addLock(
    bytes32 Id,
    bytes32 hashlock,
    uint48 timelock
  ) external _exists(Id) _validTimelock(timelock) nonReentrant returns
(bytes32) {
    HTLC storage htlc = contracts[Id];
    if (htlc.claimed == 2 || htlc.claimed == 3) revert AlreadyClaimed();
    if (msg.sender == htlc.sender) {
      if (htlc.hashlock == bytes32(bytes1(0x01))) {
        htlc.hashlock = hashlock;
        htlc.timelock = timelock;
      } else {
        revert HashlockAlreadySet(); // Prevent overwriting hashlock.
      }
      emit TokenLockAdded(Id, hashlock, timelock);
      return Id;
    } else {
      revert NoAllowance(); // Ensure only allowed accounts can add a lock.
    }
  }
```

LYSWP2-3

ERC20 DECIMAL MISMATCH MAY CAUSE INCORRECT TOKEN TRANSFERS & FUND LOSS IN CROSS-CHAIN SWAPS

SEVERITY: Informational

PATH:

chains/evm/solidity/contracts/HashedTimeLockERC20.sol#L318-L372

REMEDIATION:

LPs and users should be aware of the differences between tokens across different chains. We should include this information in the documentation notes for users.

STATUS: Fixed

DESCRIPTION:

The contract assumes a consistent token decimal structure, but ERC20 tokens have varying decimals across different chains, leading to incorrect amount transfers during cross-chain swaps.

Example:

- **USDT(USDC)** on Ethereum has **6 decimals**, while **USDT(USDC)** on BSC has **18 decimals**.
- If a user locks **1 USDT (1,000,000 units on Ethereum)** and the contract transfers it without adjusting for decimals, the recipient on BSC may receive **1,000,000 USDT instead of 1 USDT**, inflating the transfer by 1,000,000x.
- Conversely, a 1 USDT lock from BSC (1,000,000,000,000,000,000 units) could result in only 0.000000000001 USDT being claimable on Ethereum, leading to a massive loss of funds.

Allowing the sender to specify the amount in the **lock** function does not fully resolve the ERC20 decimal mismatch issue. While the function ensures that the specified amount is transferred from the sender, it does not account for differences in token decimal precision across chains. This can lead to significant discrepancies in the amounts being locked and later redeemed.

For example, if a user locks 1 USDT on Ethereum, which has 6 decimals, the contract will store the amount as 1,000,000 units. However, if the same event data is used to create a corresponding lock on BSC, where USDT has 18 decimals, the recipient may receive 1,000,000 USDT instead of 1 USDT. Conversely, if the same logic is applied in reverse, the user may receive only a fraction of the intended amount. This issue arises because the contract does not normalize amounts based on the token's decimal precision before storing them in the HTLC structure.

Another major concern is that many liquidity providers rely on emitted event data to create locks on the destination chain. Since the contract emits the locked amount without considering the token's decimal precision, LPs may use these raw values to initiate transfers on the destination chain without applying proper scaling. This can result in incorrect fund distributions, leading to potential financial losses and failed swaps.

```
  function commit(
    string[] calldata hopChains,
    string[] calldata hopAssets,
    string[] calldata hopAddresses,
    string calldata dstChain,
    string calldata dstAsset,
    string calldata dstAddress,
    string calldata srcAsset,
    bytes32 Id,
    address srcReceiver,
    uint48 timelock,
    uint256 amount,
    address tokenContract
  ) external _validTimelock(timelock) nonReentrant returns (bytes32) {
    // Ensure the generated ID does not already exist to prevent
overwriting.
    if (hasHTLC(Id)) revert HTLCAlreadyExists();
```

```
    if (amount == 0) revert FundsNotSent(); // Ensure funds are sent.
    IERC20 token = IERC20(tokenContract);

    if (token.balanceOf(msg.sender) < amount) revert InsufficientBalance();
    if (token.allowance(msg.sender, address(this)) < amount) revert
NoAllowance();
    token.safeTransferFrom(msg.sender, address(this), amount);

    // Store HTLC details.
    contracts[Id] = HTLC(
      amount,
      bytes32(bytes1(0x01)),
      uint256(1),
      tokenContract,
      timelock,
      uint8(1),
      payable(msg.sender),
      payable(srcReceiver)
    );

    // Emit the commit event.
    emit TokenCommitted(
      Id,
      hopChains,
      hopAssets,
      hopAddresses,
      dstChain,
      dstAddress,
      dstAsset,
      msg.sender,
      srcReceiver,
      srcAsset,
      amount,
      timelock,
      tokenContract
    );
    return Id;
  }
```

```
  function lock(
    bytes32 Id,
    bytes32 hashlock,
    uint256 reward,
    uint48 rewardTimelock,
    uint48 timelock,
    address srcReceiver,
    string calldata srcAsset,
    string calldata dstChain,
    string calldata dstAddress,
    string calldata dstAsset,
    uint256 amount,
    address tokenContract
  ) external _validTimelock(timelock) nonReentrant returns (bytes32) {
    if (hasHTLC(Id)) revert HTLCAlreadyExists();
    if (amount == 0) revert FundsNotSent();
    if (rewardTimelock > timelock || rewardTimelock < block.timestamp)
revert InvaliRewardTimelock();
    IERC20 token = IERC20(tokenContract);

    if (token.balanceOf(msg.sender) < amount + reward) revert
InsufficientBalance();
    if (token.allowance(msg.sender, address(this)) < amount + reward) revert
NoAllowance();

    token.safeTransferFrom(msg.sender, address(this), amount + reward);
    contracts[Id] = HTLC(
      amount,
      hashlock,
      uint256(1),
      tokenContract,
      timelock,
      uint8(1),
      payable(msg.sender),
      payable(srcReceiver)
    );

    if (reward != 0) {
      rewards[Id] = Reward(reward, rewardTimelock);
    }

    emit TokenLocked(
      Id,
      hashlock,
```

```
        dstChain,
        dstAddress,
        dstAsset,
        msg.sender,
        srcReceiver,
        srcAsset,
        amount,
        reward,
        rewardTimelock,
        timelock,
        tokenContract
    );
    return Id;
}
```

LYSWP2-10

TODO'S IN CODE

SEVERITY: Informational

PATH:

chains/starknet/src/HashTimeLockedERC20.cairo#L94
chains/starknet/src/HashTimeLockedERC20.cairo#L117
chains/starknet/src/HashTimeLockedERC20.cairo#L601
chains/starknet/src/HashTimeLockedERC20.cairo#L631

REMEDIATION:

Consider resolving the TODO's.

STATUS: Fixed

DESCRIPTION:

There are some TODO's in the code inside HashTimeLockedERC20.cairo.

```
//TODO: Check if this should be IERC20SafeDispatcher
use openzeppelin::token::erc20::interface::{IERC20Dispatcher,
IERC20DispatcherTrait};
```

```
//TODO: check if this should be public or Not?
#[derive(Drop, Serde, starknet::Store)]
pub struct HTLC {
```

```
//TODO: should this be here, as the ecdsa_signature check is already done
assert!(false, "Couldn't Recover The Public Key")
```

```
//TODO: Check if this function should be inline?
impl InternalFunctions of InternalFunctionsTrait {
```

LYSWP2-11

COMMENT IN CODE COULD BE IMPROVED

SEVERITY: Informational

PATH:

chains/starknet/src/HashTimeLockedERC20.cairo#L498-L499
chains/starknet/src/HashTimeLockedERC20.cairo#L572

REMEDIATION:

Consider revising the comments to improve readability.

Line 498:

// if the caller is the receiver then they should get the amount, and the reward

Line 572:

// update the hashlock and timelock in the storage

STATUS: Acknowledged

DESCRIPTION:

A few comments in the code inside **HashTimeLockedERC20.cairo** contract could be improved for better readability.

```
// if the caller is the receiver then they should get and the amount, and
the
// reward
```

```
// update the hashlcok and timelock in the storage
self.contracts.entry(Id).hashlock.write(hashlock);
```

INCONSISTENT VALIDATION OF REWARDTIMELOCK IN LOCK() FUNCTION BETWEEN EVM CHAINS AND STARKNET

SEVERITY: Informational

PATH:

chains/evm/solidity/contracts/HashedTimeLockERC20.sol#L334
chains/starknet/src/HashTimeLockedERC20.cairo#L409-L411

REMEDIATION:

Consider adjusting the HashTimeLockedERC20.cairo contract to validate rewardTimelock consistently inside the lock() fn.

```
assert!(
-    rewardTimelock > get_block_timestamp() && rewardTimelock <= timelock,
+    rewardTimelock >= get_block_timestamp() && rewardTimelock <= timelock,
    "Invalid Reward TimeLock"
);
```

STATUS: Fixed

DESCRIPTION:

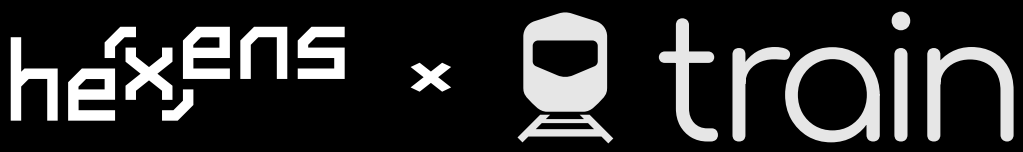
The **HashedTimeLockERC20::lock()** function permits **rewardTimelock** to be set equal to **block.timestamp** (line 334).

In contrast, **HashedTimelockERC20::lock()** enforces that **rewardTimelock** must be strictly greater than **get_block_timestamp()** (line 410), rather than allowing equality.

This inconsistency leads to a discrepancy in the lock logic between the contract implementations on the EVM chain and Starknet.

```
assert!(
    rewardTimelock > get_block_timestamp() && rewardTimelock <= timelock,
    "Invalid Reward TimeLock"
);
```

```
if (rewardTimelock > timelock || rewardTimelock < block.timestamp) revert
InvaliRewardTimelock();
    IERC20 token = IERC20(tokenContract);
```


hexens
×
train