hexens × Layerswap

DEC.24

SECURITY REVIEW REPORT FOR LAYERSWAP

CONTENTS

- About Hexens
- Executive summary
    - Scope
- Auditing details
- Severity structure
    - Severity characteristics
    - Issue symbolic codes
- Findings summary
- Weaknesses
    - Unfairly slash LP’s locked funds
    - Assets may be locked in HashedTimeLockEther and HashedTimeLockERC20 when an existing HTLC is overwritten
    - Steal LP's fund
    - Typo in Error Message ("InvalidSigniture")
    - Use custom errors
    - Unused Errors

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.

SCOPE

The analyzed resources are located on:

https://github.com/layerswap/layerswap-atomic-bridge/blob/main-audit-evm-11-29-14/chains/evm/solidity/contracts/HashedTimeLockERC20.sol

https://github.com/layerswap/layerswap-atomic-bridge/blob/main-audit-evm-11-29-14/chains/evm/solidity/contracts/HashedTimeLockEther.sol

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

https://github.com/layerswap/layerswap-atomic-bridge/commit/1ba9c9273b381e8fae1114ade1d006e31615c740

https://github.com/layerswap/layerswap-atomic-bridge/commit/6eb33021dc5d0aa59a0af478117f1f34dba13318

AUDITING DETAILS

STARTED
02.12.2024

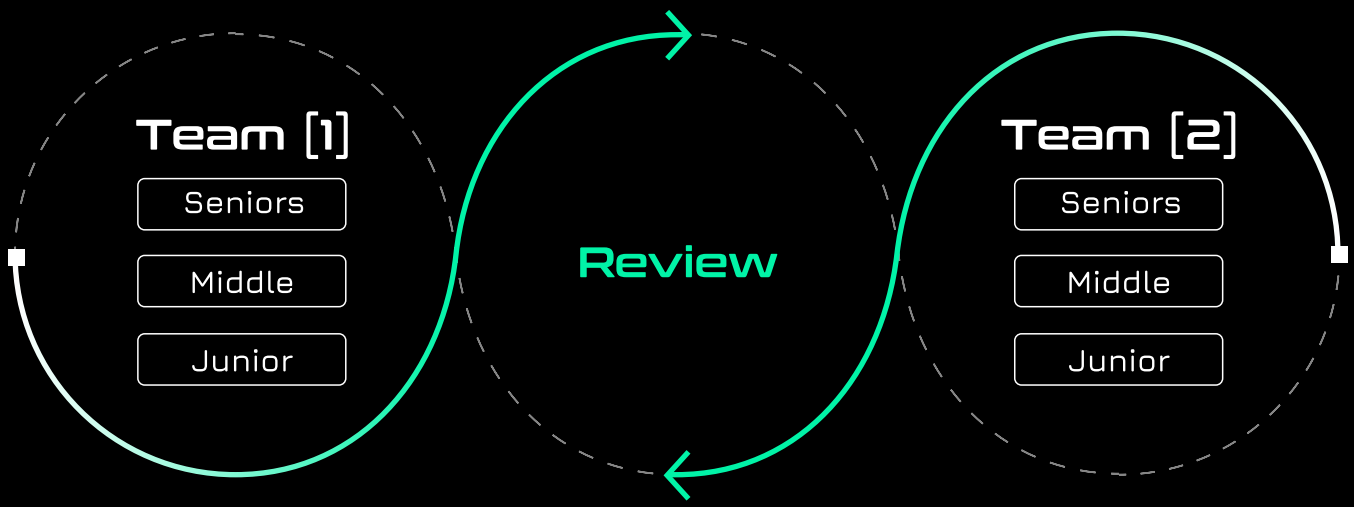
DELIVERED
09.12.2024

Review Led by

NOUREDDINE BENOMARI
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.

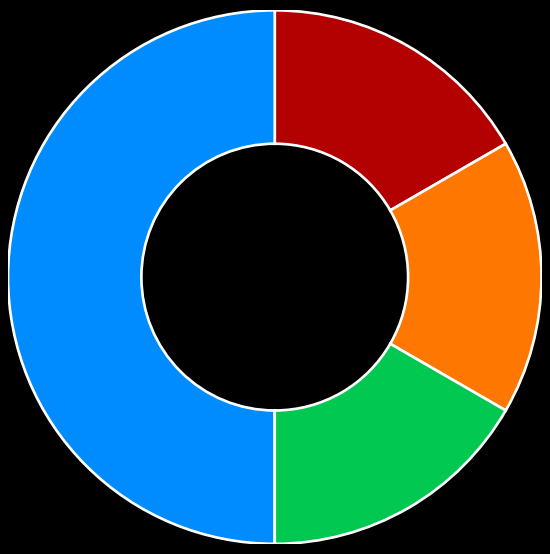
ISSUE SYMBOLIC CODES

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.

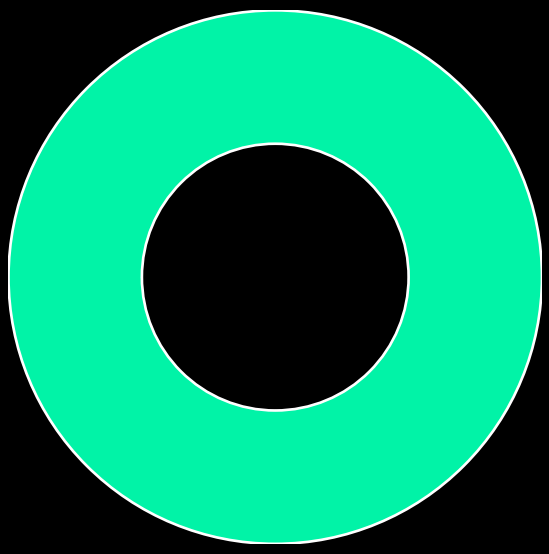
FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	0
Medium	1
Low	1
Informational	3

Total: 6



- Critical
- Medium
- Low
- Informational



- Fixed

WEAKNESSES

This section contains the list of discovered weaknesses.

LYSWP-7

UNFAIRLY SLASH LP'S LOCKED FUNDS

SEVERITY: Critical

PATH:

HashedTimeLockEther.sol#L342-L358

REMEDIATION:

Monitor whether the LP has tried to call redeem several times without success, since the LP have no interest in maliciously making this call fail, slash shouldn't be possible in this situation.

STATUS: Fixed

DESCRIPTION:

As a guarantee to ensure that LP release the user's funds, LPs can voluntarily stack a small portion of their funds in a Atomic Stake pool, if they fail to release users found this portion will be slashed. LPs with locked amounts in the atomic pool will be more favorable to users during the LP selection process.

In HashedTimeLockEther.sol, this slash can easily be forced by simply using a malicious contract that will always revert when receiving fund unless tx.origin == the attacker. This would allow him to call redeem and get his assets back in the destination chain after slashing the LP stack in the atomic pool, while preventing the LP to release the fund for the attacker.

```
function redeem(
    bytes32 Id,
    uint256 secret
) external _exists(Id) returns (bool) {
    HTLC storage htlc = contracts[Id];

    if (htlc.hashlock != sha256(abi.encodePacked(secret)))
        revert HashlockNotMatch(); // Ensure secret matches hashlock.
    if (htlc.claimed == 3 || htlc.claimed == 2) revert AlreadyClaimed();

    htlc.claimed = 3;
    htlc.secret = secret;
    (bool success, ) = htlc.srcReceiver.call{value: htlc.amount}("");
    if (!success) revert TransferFailed();
    emit TokenRedeemed(Id, msg.sender, secret, htlc.hashlock);
    return true;
}
```

ASSETS MAY BE LOCKED IN HASHEDTIMELOCKETHER AND HASHEDTIMELOCKERC20 WHEN AN EXISTING HTLC IS OVERWRITTEN

SEVERITY: Medium

PATH:

HashedTimeLockEther.sol#L169-L196

HashedTimeLockERC20.sol#L181-L218

REMEDIATION:

Reverting in the commit() function whenever an ID exists will break the commit() function because contractNonce cannot increase anymore, as ID = bytes32(blockHashAsUint ^ contractNonce).

Therefore, consider placing the Id calculation in a loop that increments contractNonce until a non-existent Id is found.

STATUS: Fixed

DESCRIPTION:

HashedTimeLockEther::commit and **HashedTimeLockERC20::commit** may overwrite an already existing HTLC (hashed time-locked contract) (line 188 in HashedTimeLockEther.sol and line 209 in HashedTimeLockERC20.sol).

Example:

1. An actor calls **HashedTimeLockEther::lock** for a specific **Id** which is then stored on line 313 into the **contracts** storage variable.
2. Later, **HashedTimeLockEther::commit** may be called where the same **Id** may be generated (line 185 HashedTimeLockEther.sol), which would then cause the protocol to overwrite (line 188 in HashedTimeLockEther.sol) the existing HTLC.

3. As a consequence funds may be lost due to this issue.

```
function commit(
        string[] calldata hopChains,
        string[] calldata hopAssets,
        string[] calldata hopAddresses,
        string calldata dstChain,
        string calldata dstAsset,
        string calldata dstAddress,
        string calldata srcAsset,
        address srcReceiver,
        uint48 timelock
    ) external payable returns (bytes32 Id) {
        if (msg.value == 0) revert FundsNotSent(); // Ensure funds are sent.
        if (timelock < block.timestamp) revert NotFutureTimelock(); //
Ensure timelock is in the future.
        unchecked {
            ++contractNonce; // Increment nonce for uniqueness.
        }
        Id = bytes32(blockHashAsUint ^ contractNonce);

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

```
function commit(
        string[] calldata hopChains,
        string[] calldata hopAssets,
        string[] calldata hopAddresses,
        string calldata dstChain,
        string calldata dstAsset,
        string calldata dstAddress,
        string calldata srcAsset,
        address srcReceiver,
        uint48 timelock,
        uint256 amount,
        address tokenContract
    ) external returns (bytes32 Id) {
        if (amount == 0) revert FundsNotSent(); // Ensure funds are sent.
        if (timelock < block.timestamp) revert NotFutureTimelock(); //
Ensure timelock is in the future.
        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);

        unchecked {
            ++contractNonce; // Increment nonce for uniqueness.
        }
        Id = bytes32(blockHashAsUint ^ contractNonce);

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

LYSWP-8

STEAL LP'S FUND

SEVERITY: Low

PATH:

HashedTimeLockEther.sol#L237-L257

REMEDIATION:

Force a minimum timelock value in the addLock() function.

STATUS: Fixed

DESCRIPTION:

In the source chain, the user have a total control on the timelock he will set for his funds. This allow him to steal from the LP by using the refund function while still redeeming on the destination chain, here are the steps of how the attack would go :

Steps to Exploit

1. User initiates a cross-chain swap:
   - The user calls **commit** on the source chain.
   - The LP locks funds on the destination chain by calling **lock**.
2. User sets a minimal timelock:
   - The user calls **addLock** on the source chain with:
     - A **timelock** set to **block.timestamp** + 1.
     - The generated hashlock.
3. User front-runs LP's redeem attempt:
   - On the next block, the LP attempts to redeem the funds.
   - The user front-runs this transaction and calls **refund**:
     - Since the timelock (**block.timestamp + 1**) has already passed, the refund succeeds.
   - The user obtains the secret from the **hashlock** in the process.

4. User redeems on the destination chain:
    - Using the secret, the user calls **redeem** on the destination chain before the LP's timelock expires (usually set to ~15 minutes).
    - The user successfully retrieves the funds on the destination chain.
5. Optional exploitation with Atomic Pool:
    - If the LP uses an atomic pool, the user can additionally call **slash** to penalize the LP further by slashing its stake.

Consequences : the attacker get his refund on source chain AND get the fund on destination chain, while slashing the LP stacked fund in the process too.

```
function addLock(
        bytes32 Id,
        bytes32 hashlock,
        uint48 timelock
    ) external _exists(Id) returns (bytes32) {
        HTLC storage htlc = contracts[Id];
        if (htlc.claimed == 2 || htlc.claimed == 3) revert AlreadyClaimed();
        if (timelock < block.timestamp) revert NotFutureTimelock();
        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.
        }
    }
```

TYPO IN ERROR MESSAGE ("INVALIDSIGNITURE")

SEVERITY: Informational

PATH:

HashedTimeLockERC20.sol#L57

HashedTimeLockEther.sol#L55

LayerswapV8.sol#L189

REMEDIATION:

Correct the typo.

```
-- revert InvalidSigniture(); // Ensure valid signature.
++ revert InvalidSignature(); // Ensure valid signature.
```

STATUS: Fixed

DESCRIPTION:

In the error message **InvalidSigniture();**, the word "**Signature**" is spelled incorrectly as "**Signiture**." It should be spelled "**Signature**."

LYSWP-5

USE CUSTOM ERRORS

SEVERITY: Informational

PATH:

HashedTimeLockERC20.sol#L162-L165

STATUS: Fixed

DESCRIPTION:

Custom Errors, available from Solidity compiler version 0.8.4, provide benefits such as smaller contract size, improved gas efficiency, and better protocol interoperability. Replace **require** statements with Custom Errors for a more streamlined and user-friendly experience. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

```
modifier _exists(bytes32 Id) {
    require(hasHTLC(Id), "HTLC Not Exists");
    _;
}
```

UNUSED ERRORS

SEVERITY: Informational

PATH:

HashedTimeLockERC20.sol#L53, L59

REMEDIATION:

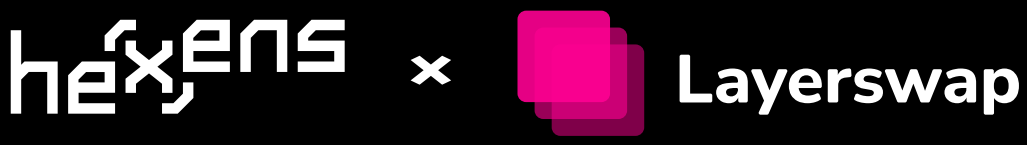
If the error is redundant and there is no missing logic where it can be used, it should be removed.

STATUS: Fixed

DESCRIPTION:

In the **LayerswapV8ERC20** contract, the errors **HTLCNotExists** and **TransferFailed** on line (L53, L59) are declared but never used.

```
    /// @notice Amounts already withdrawn this period for each token.
mapping(address => uint256) public rateLimitDuration;
```


hexens
×
Layerswap