hexens × polygon

Oct.23

SECURITY REVIEW REPORT FOR POLYGON TECHNOLOGY

CONFIDENTIAL

CONFIDENTIAL

CONTENTS

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.

CONFIDENTIAL

AUDIT LED BY

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

Audit Starting Date
16.10.2023

Audit Completion Date
23.10.2023

---

hexens × polygon

CONFIDENTIAL

LIMITATIONS ON DISCLOSURE AND USAGE OF THIS REPORT

This report has been developed by the company Hexens (the Service Provider) based on the Smart Contract Audit of Polygon Technology (the Client). The document contains vulnerability information and remediation advice.

The information, presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Polygon Technology.

If you are not the intended recipient of this document, remember that any disclosure, copying or dissemination of it is forbidden.

METHODOLOGY

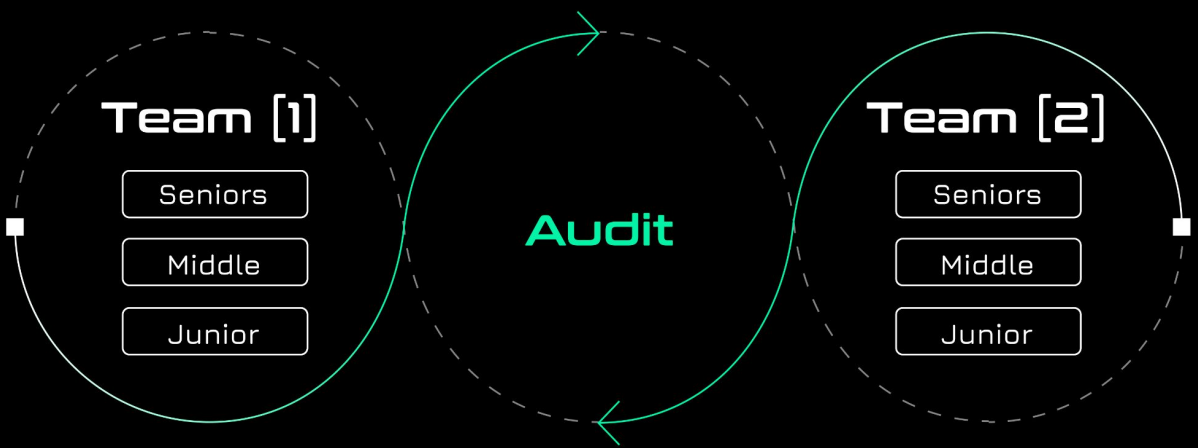
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor* Audit

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

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 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.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

SCOPE

The analyzed resources are located on:
https://github.com/omnifient/usdc-e
https://github.com/omnifient/usdc-lxly
https://github.com/pyk/zkevm-wsteth

CONFIDENTIAL

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	0
MEDIUM	0
LOW	1
INFORMATIONAL	2

TOTAL: 3

SEVERITY

STATUS

WEAKNESSES

This section contains the list of discovered weaknesses.

PUSD-3. POSSIBILITY OF LIQUIDITY ATTACK

SEVERITY: Low

PATH: usdc-lxly/src/NativeConverter.sol

REMEDIATION: in order to maintain a healthy liquidity balance, migrate should be called often through some automated service or potentially in the convert function after the pending amount exceeds some threshold (so small conversions won't pay too much gas)

STATUS:

DESCRIPTION:

The USDC LXLY bridge allows for USDC from mainnet to be bridged to Polygon zkEVM as USDCe. It also implements a NativeConverter contract where users can hand in their zkBWUSDC and obtain newly minted USDCe.

The contract also exposes the **migrate** function, which permisionlessly sends the zkBWUSDC back to the L1Escrow contract through the **zkEVMBridge**. The L1Escrow contract would then receive USDC, so it can supply this to users in case any USDCe minted from conversion is redeemed.

Normally, a bridge would have the exact amount of tokens locked equal to the total minted amount of the token on L2. E.g., there would be the same amount of USDC locked in L1Escrow as USDCe minted on L2.

However, through the NativeConverter, it becomes possible to increase the total amount of USDCe arbitrarily and at little cost. This USDC will be unaccounted for on L1, until someone calls **migrate** and the bridging is completed. This creates the possibility for malicious users to temporarily drain L1 of USDC liquidity by repeatedly converting and bridging.

```
function convert(
    address receiver,
    uint256 amount,
    bytes calldata permitData
) external whenNotPaused {
    require(receiver != address(0), "INVALID_RECEIVER");
    require(amount > 0, "INVALID_AMOUNT");

    if (permitData.length > 0)
        LibPermit.permit(address(zkBWUSDC), amount, permitData);

    // transfer the wrapped usdc to the converter, and mint back native usdc
    zkBWUSDC.safeTransferFrom(msg.sender, address(this), amount);
    zkUSDCe.mint(receiver, amount);

    emit Convert(msg.sender, receiver, amount);
}

function migrate() external whenNotPaused {
    // Anyone can call migrate() on NativeConverter to
    // have all zkBridgeWrappedUSDC withdrawn via the PolygonZkEVMBridge
    // moving the L1_USDC held in the PolygonZkEVMBridge to L1Escrow

    uint256 amount = zkBWUSDC.balanceOf(address(this));

    if (amount > 0) {
        zkBWUSDC.approve(address(bridge), amount);

        bridge.bridgeAsset(
            l1NetworkId,
            l1Escrow,
            amount,
            address(zkBWUSDC),
            true, // forceUpdateGlobalExitRoot
            "" // empty permitData because we're doing approve
        );

        emit Migrate(amount);
    }
}
```

PUSD-1. UNUSED IMPORTS IN POLYGONBRIDGELIBUPGRADEABLE CONTRACT

SEVERITY: Informational

PATH: base/PolygonBridgeLibUpgradeable.sol

REMEDIATION: remove the unused imports from the PolygonBridgeLibUpgradeable contract

STATUS:

DESCRIPTION:

The **PolygonBridgeLibUpgradeable** contract has two imports: **import "../interfaces/IBasePolygonZkEVMGlobalExitRoot.sol"**; (L7) and **import "../interfaces/IBridgeMessageReceiver.sol";** (L8) that are likely unused in the current version of the contract.

```
import "../interfaces/IBasePolygonZkEVMGlobalExitRoot.sol";
import "../interfaces/IBridgeMessageReceiver.sol";
```

PUSD-2. DEFAULT VALUE INITIALISATION

SEVERITY: Informational

PATH: Vsrc/WstETHBridgeLX.sol

REMEDIATION: remove the initialisation part of the variable declaration

STATUS:

DESCRIPTION:

In **WstETHBridgeL1.sol** and **WstETHBridgeL2.sol**, the storage variable **originTokenNetwork** is directly initalised to its default value **0**.

This is unnecessary because in Solidity any storage variable will have this value by default and setting it again will result in a waste of gas.

```
uint32 public originTokenNetwork = 0;
```