hexens × polygon

MAY.24

SECURITY REVIEW REPORT FOR POLYGON

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.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered an extension called Bridge and Call smart contract for the LxLy Bridge, part of the Polygon blockchain. It is a smart contract that allows users to invoke a single call to both bridge an asset and send a message. To fulfill this purpose, the extension implements a Jump point contract, which executes the message whenever the tokens are first bridged to this contract.

Our security assessment was a full review of the Bridge and Call extension, spanning a total of 3 days.

During our audit, we identified one critical severity vulnerability. The vulnerability could result in users losing their tokens if the WETH hasn't been initialized in the bridge contract.

We also identified one high and one medium issue, along with various low vulnerabilities.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/AggLayer/lxly-bridge-and-call

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

https://github.com/AggLayer/lxly-bridge-and-call/commit/fed2b23e557d2a5fca040993d10b18590351b608

AUDITING DETAILS

STARTED
27.05.2024

DELIVERED
30.05.2024

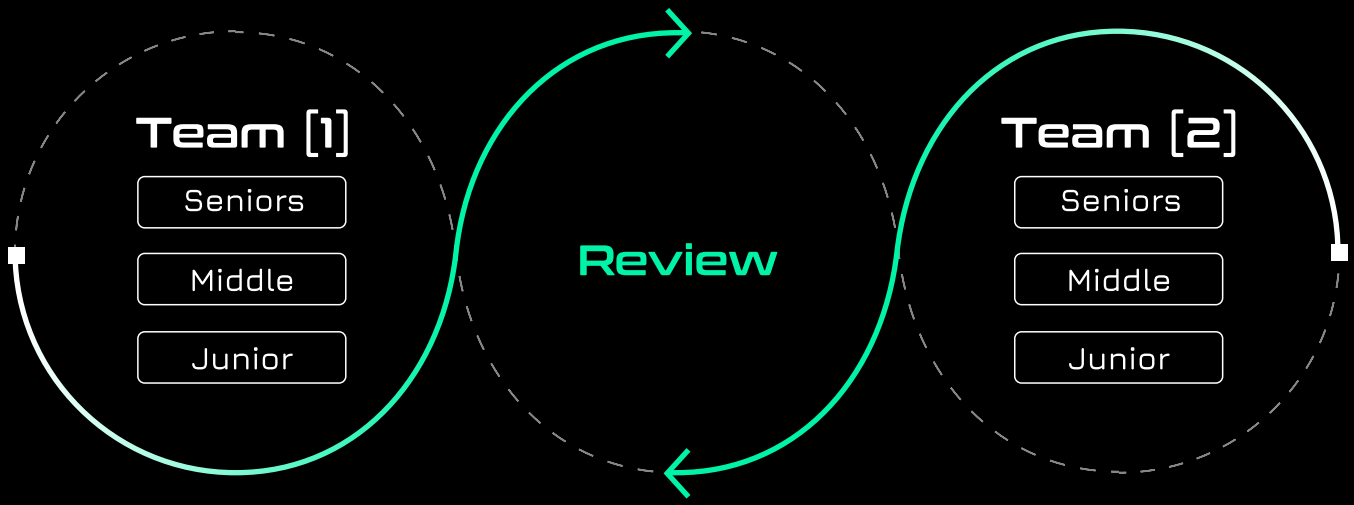
Review Led by

TRUNG DINH

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.

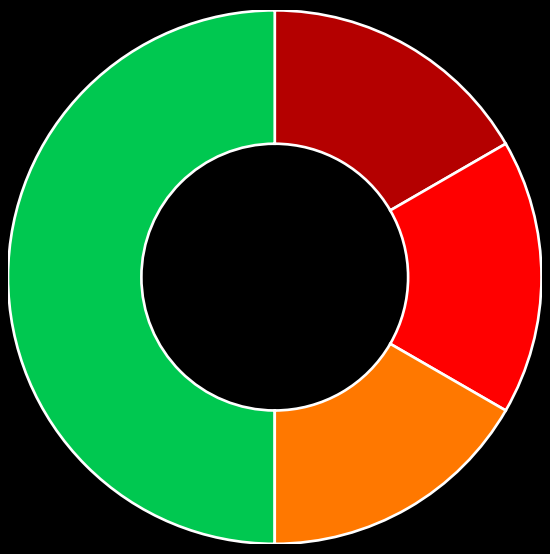
ISSUE SYMBOLIC CODES

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.

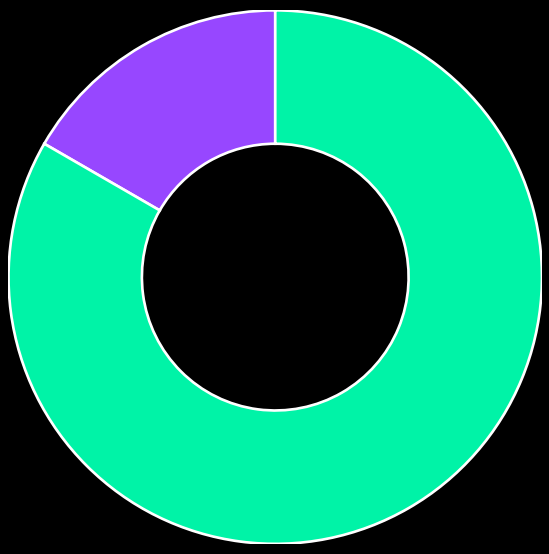
FINDINGS SUMMARY

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

Total: 6



- Critical
- High
- Medium
- Low



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

PAGL-3

LOSS OF FUNDS DUE TO INCORRECT ENCODEDMSG IN BRIDGEEXTENSION.BRIDGEANDCALL() FUNCTION

SEVERITY: Critical

PATH:

WETHPlus.sol:supportsInterface():L75-80

REMEDIATION:

Consider modifying the line 78 to:

```
- if (token == address(bridge.WETHToken())) {
+ if (token != address(0) && token == address(bridge.WETHToken())) {
```

STATUS: Fixed

DESCRIPTION:

The issue is about to occur in the network such that **bridge.WETHToken = address(0)**. Assume this happens and a user wants to trigger the function **BridgeExtension.bridgeAndCall()** with the input **token = address(0)**.

The **bridgeAndCall()** function can be separated into two steps: bridge asset and bridge message. Applying the scenario above to these two steps, we have:

1. Bridge Asset

   Since **bridge.WETHToken = token = address(0)**, the function will process the if block from line 55 to line 62, which invokes the internal function **_bridgeNativeAssetHelper()** to bridge the asset. Within this internal function, the **jumpPointAddress** is calculated by using:

   - **assetNetwork = bridge.gasTokenNetwork()**
   - **assetAddress = bridge.gasTokenAddress()**

2. Bridge Message

   Since **token = bridge.WETHToken** (both are **address(0)**), the first if block from line 78 to line 80 will be processed. This if block encodes **bridge.networkID()** and **address(0)** into **encodedMessage**. These two parameters will then be decoded into:

   - **assetOriginalNetwork = bridge.networkID()**
   - **assetOriginalAddress = address(0)**

   in the **onMessageReceived()** on the destination chain. These two parameters will then be used within the constructor for the new **JumpPoint** contract, which consequently generates another **JumpPoint** contract different from the **jumpPointAddress** calculated in step 1.

Due to the inconsistency in using the **JumpPoint** contract address in these two steps, the tokens from users will be lost.

src/BridgeExtension.sol:L55-L62

```
} else if (token == address(0)) {
    // user is bridging the gas token
    if (msg.value != amount) {
        revert AmountDoesNotMatchMsgValue();
    }


    // transfer native gas token (e.g. eth) - using a helper to get rid of
stack too deep
        _bridgeNativeAssetHelper(amount, destinationNetwork, callAddress,
fallbackAddress, callData, dependsOnIndex);
```

src/BridgeExtension.sol:L78-L80

```
if (token == address(bridge.WETHToken())) {
    encodedMsg =
                abi.encode(dependsOnIndex, callAddress, fallbackAddress,
bridge.networkID(), address(0), callData);
```

FUNDS WILL BE LOST WHEN RECEIVING USDT OR OTHER TOKENS ON THE MAINNET BECAUSE OF THE NON-COMPLIANT ERC20 INTERFACE

SEVERITY: High

PATH:

src/JumpPoint.sol:L69-L70

REMEDIATION:

Use the SafeERC20 library.

STATUS: Fixed

DESCRIPTION:

The ERC20 interface used in the JumpPoint contract is the standard **IERC20** from OpenZeppelin. The interface specifies the **transfer()** function as:

```
function transfer(address to, uint256 value) external returns (bool);
```

The problem is that some tokens on the mainnet, such as USDT or BNB (more), are not fully ERC20 compliant, and their transfer interface doesn't return the boolean:

```
function transfer(address to, uint256 value) external;
```

Solidity automatically implements checks of the return value and reverts if it's absent.

As a result, when users receive funds through the extension, and the call to callAddress is not successful for some reason, the transaction will always halt on the transfer to **fallbackAddress** and the funds will be stuck.

```
// if call was unsuccessful, then transfer the asset to the fallback address
if (!success) asset.transfer(fallbackAddress, balance);
```

PAGL-1

BRIDGEEXTENSION DOESN'T SUPPORT TOKENS WITH AMOUNT CHANGES IN TRANSFERS (LIKE FEE-ON-TRANSFER)

SEVERITY: Medium

PATH:

src/BridgeExtension.sol:L49

src/BridgeExtension.sol:L66

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

BridgeExtension.bridgeAndCall() reverts if there's a change in the amount on one of the ERC20 transfers because the actual amount received can be less than the specified in the **amount** variable.

When the incorrect **amount** is passed to **bridge.bridgeAsset()**, the transaction will halt because of the insufficient balance.

If the actual amount received is larger than the specified in the **amount** variable, the transaction will pass, but the user will lose some funds.

```
IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
```

Use the difference between balances like you do in the bridge:

zkevm-contracts/contracts/v2/PolygonZkEVMBridgeV2.sol at f7d97e315a2692d5806ea69be09c93490bbda617 · 0xPolygonHermez/zkevm-contracts

```
// In order to support fee tokens check the amount received, not the transferred
uint256 balanceBefore = IERC20Upgradeable(token).balanceOf(
    address(this)
);
IERC20Upgradeable(token).safeTransferFrom(
    msg.sender,
    address(this),
    amount
);
uint256 balanceAfter = IERC20Upgradeable(token).balanceOf(
    address(this)
);
```

PAGL-4

UNUSED PERMITDATA PARAMETER IN FUNCTION POLYGONZKEVMBRIDGEV2.BRIDGEASSET()

SEVERITY: Low

PATH:

src/BridgeExtension.sol:L36

REMEDIATION:

Consider removing the permitData out of the input parameters of function bridgeAndCall().

If the permit is required within the function, we recommend triggering the permit() function before line 66 to help the contract BridgeExtension pull tokens from the sender.

STATUS: Fixed

DESCRIPTION:

Within the function **PolygonZkEVMBridgeV2.bridgeAsset()**, **permitData** is used to grant the allowance from the sender to the bridge contract, allowing the bridge to pull tokens from the **msg.sender** to execute the action.

zkevm-contracts/contracts/v2/PolygonZkEVMBridgeV2.sol at 1ad7089d04910c319a257ff4f3674ffd6fc6e64e · 0xPolygonHermez/zkevm-contracts

```
// Use permit if any
function bridgeAsset(
    uint32 destinationNetwork,
    address destinationAddress,
    uint256 amount,
    address token,
    bool forceUpdateGlobalExitRoot,
    bytes calldata permitData
) public payable virtual ifNotEmergencyState nonReentrant {
    ...

    if (permitData.length != 0) {
        _permit(token, amount, permitData);
    }

    // To support fee tokens, check the amount received, not the transferred
amount
                                    uint256         balanceBefore         =
IERC20Upgradeable(token).balanceOf(address(this));
        IERC20Upgradeable(token).safeTransferFrom(msg.sender,  address(this),
amount);

    ...
}
```

However, in the function **BridgeExtension.bridgeAndCall()**, the allowance for the token is already given to the bridge contract at line 176 before calling **bridge.bridgeAsset()**. Therefore, permitData is not needed in the **bridgeAndCall()** function.

```
function bridgeAndCall(
    address token,
    uint256 amount,
    bytes calldata permitData,
    uint32 destinationNetwork,
    address callAddress,
    address fallbackAddress,
    bytes calldata callData,
    bool forceUpdateGlobalExitRoot
) external payable {
```

PAGL-5

VARIABLE BRIDGE CAN BE SET TO IMMUTABLE

SEVERITY: Low

PATH:

src/BridgeExtension.sol:L22

REMEDIATION:

Consider changing the bridge variable to immutable.

STATUS: Acknowledged

DESCRIPTION:

The address variable **bridge** is only set once during the **initialize()** function, and there is no setter function to modify its value. Therefore, we can declare bridge as an immutable variable to save gas.

```
PolygonZkEVMBridgeV2 public bridge;
```

REDUNDANCY IN APPROVING WRAPPED TOKENS FOR THE BRIDGE CONTRACT

SEVERITY: Low

PATH:

src/BridgeExtension.sol#L175-L179

src/BridgeExtension.sol#L124-L128

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Within the function **PolygonZkEVMBridgeV2.bridgeAsset()**, if the **token** is either **WETHToken** or a **TokenWrapped** contract, the function will burn the **token** directly from the **msg.sender**'s wallet without requiring an allowance from the sender.

https://github.com/0xPolygonHermez/zkevm-contracts/blob/1ad7089d04910c319a257ff4f3674ffd6fc6e64e/contracts/lib/TokenWrapped.sol#L60-L63

```
// Notice that is not require to approve wrapped tokens to use the bridge
function burn(address account, uint256 value) external onlyBridge {
    _burn(account, value);
}
```

In the function **BridgeExtension.bridgeAndCall()**, before triggering **bridge.bridgeAsset()**, the function always calls **approve()** on the token to the **bridge** contract. This behavior is gas-wasting since we can skip the **approve** invocation if the token is a **TokenWrapped** contract.

```
// allow the bridge to take the assets
IERC20(token).approve(address(bridge), amount);


// bridge the ERC20 assets
bridge.bridgeAsset(destinationNetwork, jumpPointAddr, amount, token, false,
permitData);
```

Consider removing line 125 of the function _bridgeNativeWETHAssetHelper() and modifying the function _bridgeERC20AssetHelper() as follows:

```
function _bridgeERC20AssetHelper(
    address token,
    uint256 amount,
    bytes calldata permitData,
    uint32 destinationNetwork,
    address callAddress,
    address fallbackAddress,
    bytes calldata callData,
    uint256 dependsOnIndex
) internal {
    address jumpPointAddr;


    {
        // we need to encode the correct token network/address
                (uint32 assetOriginalNetwork, address assetOriginalAddr) =
bridge.wrappedTokenToTokenInfo(token);
        if (assetOriginalAddr == address(0)) {
            // only do this when the token is not from this network
            assetOriginalNetwork = bridge.networkID();
            assetOriginalAddr = token;

+             IERC20(token).approve(address(bridge), amount);
        }


          // pre-compute the address of the JumpPoint contract so we can
bridge the assets
        jumpPointAddr = _computeJumpPointAddress(
                    dependsOnIndex, assetOriginalNetwork, assetOriginalAddr,
callAddress, fallbackAddress, callData
        );
    }
```

```
    // allow the bridge to take the assets
-    IERC20(token).approve(address(bridge), amount);

    // bridge the ERC20 assets
      bridge.bridgeAsset(destinationNetwork, jumpPointAddr, amount, token,
false, permitData);
}
```


hexens
×
polygon