Competitive Security Assessment

Merlin_BTC_L2

Jan 23rd, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
• Verify the code base is compliant with the most up-to-date industry standards and security best practices.
• Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

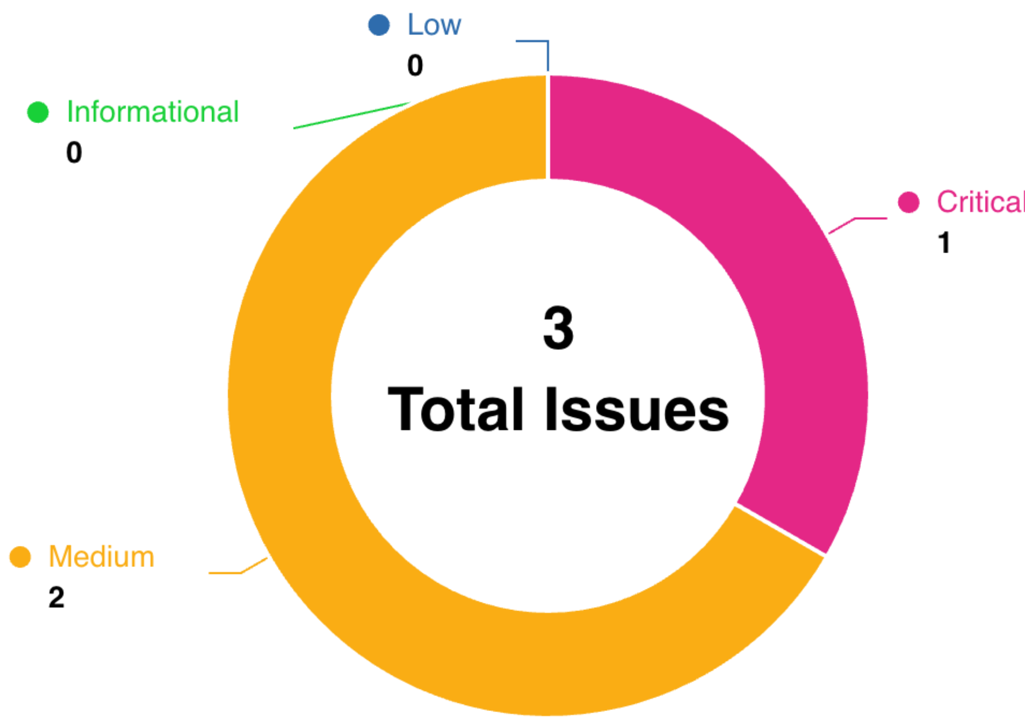
Project Detail

Project Name	Merlin_BTC_L2
Platform & Language	Solidity
Codebase	• https://github.com/MerlinLayer2/merlin-cdk-validium-contracts • audit commit - e803166f59cdb6fd99bb27abfd4d2b4d2477ea9d • final commit - 4b39836d3a19d90b291b6d9eb46753600ff4d502
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
./contracts/CDKValidium.sol	708c11b6182ff0e74ecc820a938f7792b7df6d67c2f3ed772ec3651d3aa753fb
./contracts/verifiers/FflonkVerifier.sol	7b3d7f5eb4dad7c35a3673ec3e0059509918dd827b7120ea3d669a118bd692bc
./contracts/PolygonZkEVMBridge.sol	365ef28464b92bf0f72b6bf08089b26e1eec315b858492f1a234be283754d04d
./contracts/CDKDataCommittee.sol	5e344883976750692c04d92f05f6a984262902d879db45478a0ef9507771f88d
./contracts/lib/TokenWrapped.sol	421a434bb0b24efa710e151aeb60623f4482369562933e7beaedd395f9216bdf
./contracts/lib/DepositContract.sol	a807f752e1297a2e2b7ade217975584e116f30acaba794980ebb47afdf428ea2
./contracts/PolygonZkEVMGlobalExitRoot.sol	8b002ff5177c31dc39ca9e3daffaf818163d17d57f8abdeccd847e35d401a48a
./contracts/deployment/CDKValidiumDeployer.sol	58914a665778cdd97cfc4f7fc6f562a536a9f88ac8fba70de40652c243996630
./contracts/lib/EmergencyManager.sol	0d30c56c0f7a27f5f8f69fe40322c2f25e896b00159153682a8fc75a509dcd89
./contracts/CDKValidiumTimelock.sol	b94238851a67a493f8367fa16d421d7ea8071f75c3de0a98193f733ce08a431f
./contracts/PolygonZkEVMGlobalExitRootL2.sol	aa1c6879c6ff53b654c8400c7198efc8a60efd9a3b041fb71fd9c11cd892f123
./contracts/lib/GlobalExitRootLib.sol	6f880c1ffeab850e046488ab7fd45379ca628367b335c699a5c0906d01b6c9d1

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
MBL-1	The Merkle tree branch update occurs after reaching `_MAX_DEPOSIT_COUNT` limit	Logical	Critical	Fixed	ethprinter
MBL-2	The smart contract's multi-signature verification is subject to duplicate signature vulnerability	Signature Forgery or Replay	Medium	Acknowledged	toffee
MBL-3	Deflationary token Vulnerability in bridgeAsset Function	Logical	Medium	Acknowledged	zigzag

MBL-1:The Merkle tree branch update occurs after reaching `_MAX_DEPOSIT_COUNT` limit

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	ethprinter

Code Reference

- code/contracts/lib/DepositContract.sol#L65-L89

```
65:function _deposit(bytes32 leafHash) internal {
66:        bytes32 node = leafHash;
67:
68:        // Avoid overflowing the Merkle tree (and prevent edge case in computing `_branch`)
69:        if (depositCount >= _MAX_DEPOSIT_COUNT) {
70:            revert MerkleTreeFull();
71:        }
72:
73:        // Add deposit data root to Merkle tree (update a single `_branch` node)
74:        uint256 size = ++depositCount;
75:        for (
76:            uint256 height = 0;
77:            height < _DEPOSIT_CONTRACT_TREE_DEPTH;
78:            height++
79:        ) {
80:            if (((size >> height) & 1) == 1) {
81:                _branch[height] = node;
82:                return;
83:            }
84:            node = keccak256(abi.encodePacked(_branch[height], node));
85:        }
86:        // As the loop should always end prematurely with the `return` statement,
87:        // this code should be unreachable. We assert `false` just to be safe.
88:        assert(false);
89:    }
```

Description

ethprinter : In the `DepositContract`::`_deposit()`, it is designed to update a Merkle tree branch when a new leaf node is added. But the problem is that after the `depositCount` reaches its maximum value (`_MAX_DEPOSIT_COUNT`) the Merkle tree branch is still updated because of the post-increment of `depositCount` (`++depositCount`). This will result in an inconsistent tree state and could cause unexpected results

Recommendation

ethprinter : It is recommended to modify the `depositCount` incrementation to happen only after the Merkle tree branch update has been successfully carried out and verified.

```
function _deposit(bytes32 leafHash) internal {
    bytes32 node = leafHash;

    if (depositCount >= _MAX_DEPOSIT_COUNT) {
        revert MerkleTreeFull();
    }

    // Update Merkle tree branch
    uint256 size = depositCount;
    for (uint256 height = 0; height < _DEPOSIT_CONTRACT_TREE_DEPTH; height++) {
        if (((size >> height) & 1) == 1) {
            _branch[height] = node;
            break;
        }
        node = keccak256(abi.encodePacked(_branch[height], node));
    }

    depositCount++;
}
```

Client Response

Fixed, 4b39836d3a19d90b291b6d9eb46753600ff4d502

MBL-2:The smart contract's multi-signature verification is subject to duplicate signature vulnerability

Category	Severity	Client Response	Contributor
Signature Forgery or Replay	Medium	Acknowledged	toffee

Code Reference

- code/contracts/CDKDataCommittee.sol#L103-L106

```
103:function verifySignatures(
104:        bytes32 signedHash,
105:        bytes calldata signaturesAndAddrs
106:    ) external view {
```

Description

toffee : the function `CDKDataCommittee::verifySignatures` is vulnerable for multiple identical signatures, this can allow governance cheating to bypass the `requiredAmountOfSignatures` check.

Recommendation

toffee : To reduce the risk of multiple identical signatures being submitted, the contract can keep mapping to keep track of the signature status and only add the signature to the contract if it is a new unique one prior to checking it is part of the committee

Client Response

Acknowledged

MBL-3: Deflationary token Vulnerability in bridgeAsset Function

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	zigzag

Code Reference

- code/contracts/PolygonZkEVMBridge.sol#L174-L300

```
function bridgeAsset(
        uint32 destinationNetwork,
        address destinationAddress,
        uint256 amount,
        address token,
        bool forceUpdateGlobalExitRoot,
        bytes calldata permitData
    ) public payable virtual ifNotEmergencyState nonReentrant {
        if (
            destinationNetwork == networkID ||
            destinationNetwork >= _CURRENT_SUPPORTED_NETWORKS
        ) {
            revert DestinationNetworkInvalid();
        }

        address originTokenAddress;
        uint32 originNetwork;
        bytes memory metadata;
        uint256 leafAmount = amount;

        if (token == address(0)) {
            // Ether transfer
            if ((msg.value - bridgeFee) != amount) {
                revert AmountDoesNotMatchMsgValue();
            }

            // Ether is treated as ether from mainnet
            originNetwork = _MAINNET_NETWORK_ID;
        } else {
            // Check whether msg.value is equal to the cross-chain handling fee
            if (msg.value != bridgeFee) {
                revert AmountDoesNotMatchMsgValue();
            }

            TokenInformation memory tokenInfo = wrappedTokenToTokenInfo[token];

            if (tokenInfo.originTokenAddress != address(0)) {
                // The token is a wrapped token from another network

                // Burn tokens
                TokenWrapped(token).burn(msg.sender, amount);

```

```
                originTokenAddress = tokenInfo.originTokenAddress;
                originNetwork = tokenInfo.originNetwork;
            } else {
                // 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)
                );

                // Override leafAmount with the received amount
                leafAmount = balanceAfter - balanceBefore;

                originTokenAddress = token;
                originNetwork = networkID;

                // Encode metadata
                metadata = abi.encode(
                    _safeName(token),
                    _safeSymbol(token),
                    _safeDecimals(token)
                );
            }
        }

        if (gasTokenAddress != address (0)) { // is gas token
            if (token == address(0)) {
                originTokenAddress = gasTokenAddress;
                metadata = gasTokenMetadata;
                if (networkID != _MAINNET_NETWORK_ID) { // is l2 -> l1,
                    leafAmount /= gasTokenDecimalDiffFactor;
                    if (leafAmount == 0) {
                        revert AmountTooSmall();
                    }
                }

```

```
            } else if (originTokenAddress == gasTokenAddress) {
                originTokenAddress = address(0);
                if (networkID == _MAINNET_NETWORK_ID) { // is l1 -> l2
                    leafAmount *= gasTokenDecimalDiffFactor;
                }
            }
        }

        emit BridgeEvent(
            _LEAF_TYPE_ASSET,
            originNetwork,
            originTokenAddress,
            destinationNetwork,
            destinationAddress,
            leafAmount,
            metadata,
            uint32(depositCount)
        );

        _deposit(
            getLeafValue(
                _LEAF_TYPE_ASSET,
                originNetwork,
                originTokenAddress,
                destinationNetwork,
                destinationAddress,
                leafAmount,
                keccak256(metadata)
            )
        );

        if (feeAddress != address(0) && bridgeFee > 0) {
            (bool success, ) = feeAddress.call{value: bridgeFee}(new bytes(0));
            if (!success) {
                revert EtherTransferFailed();
            }
        }

        // Update the new root to the global exit root manager if set by the user
        if (forceUpdateGlobalExitRoot) {
            _updateGlobalExitRoot();
        }
    }
```

Description

zigzag : A serious security risk involving ERC20 token transfers via a cross-chain bridge mechanism is presented by the discovered weakness in the bridgeAsset function of the supplied smart contract. In particular, this vulnerability results from improper management of ERC20 tokens that use deflationary or transfer fee mechanisms.By comparing the balances before and after the safeTransferFrom call, the contract calculates the amount of ERC20 tokens received (leafAmount). ERC20 tokens that burn a portion of their token supply during transfers or impose fees are not taken into consideration by this method.

Recommendation

zigzag : 1. Establish a system that will reliably confirm the precise quantity of ERC20 tokens that were received after the transfer. This can entail checking the contract's token balance before and after the transfer, then validating the anticipated balance decrease. 2. Include checks to make sure the contract can appropriately handle coins with deflationary characteristics or transfer fees. This might entail adding a way to query the burn rate or transfer fee of the token and modifying the computations accordingly.

Client Response

Acknowledged

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.