Zellic

Scroll

Smart Contract Security Assessment

September 7, 2023

Prepared for:

Haichen Shen

Scroll Tech

Prepared by:

Ayaz Mammadov and Vlad Toie

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Scroll Tech from July 10th to August 4th, 2023. During this engagement, Zellic reviewed Scroll's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- How does Scroll validate the transfer of tokens between the two layers?
- Are there any potential exploits in the design of each Messenger contract that could result in fund theft?
- Could there be any vulnerabilities in the verification of the proofs generated by the off-chain proof generation process?
- Can fake tokens be supplied for legitimate other-layer tokens?
- Are there any potential implications of the updates to the contracts since the last audit?
- How do the changes to the contracts affect the security of the system overall?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project, especially the off-chain components that may be relied upon to validate transactions
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

During this assessment, extensively focusing on the additions since the previous audit(May 26, 2023) prevented us from additionally modeling the potential implications of the project's off-chain components. We recommend that the Scroll Tech team consider the potential implications of the off-chain components in conjunction with the on-chain components.

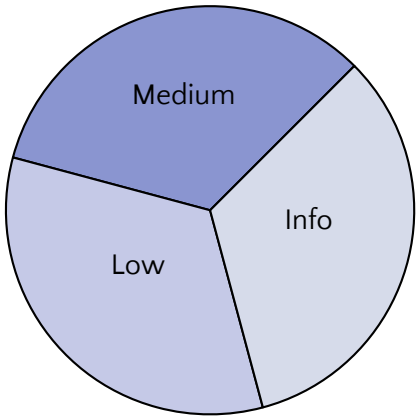
1.3 Results

During our assessment on the scoped Scroll contracts, we discovered three findings. No critical issues were found. One was of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Scroll Tech's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	1



2 Introduction

2.1 About Scroll

Scroll is a zkEVM-based zkRollup on Ethereum that enables native compatibility for existing Ethereum applications and tools.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas

optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

Scroll Contracts

Repository	https://github.com/scroll-tech/scroll
Version	scroll: `2323dd0daa117bc9ce1f0e4a1c1c3bba4d661947`

Programs	GasSwap.sol L1ScrollMessenger.sol EnforcedTxGateway.sol L1CustomERC20Gateway.sol L1ERC20Gateway.sol L1ERC721Gateway.sol L1GatewayRouter.sol L1StandardERC20Gateway.sol L1USDCGateway.sol L1WETHGateway.sol L1MessageQueue.sol L2GasPriceOracle.sol MultipleVersionRollupVerifier.sol ScrollChain.sol L2ScrollMessenger.sol L2StandardERC20Gateway.sol L2USDCGateway.sol BatchHeaderV0Codec.sol ChunkCodec.sol ScrollGatewayBase.sol RollupVerifier.sol WithdrawTrieVerifier.sol
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two and a half person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

July 10, 2023	Kick-off call
July 10, 2023	Start of primary review period
August 4, 2023	End of primary review period

3 Detailed Findings

3.1 The `onlyOwner` modifier is missing in the ScrollChain contract

- **Target**: ScrollChain
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

In the ScrollChain contract, the `importGenesisBatch` function works as an initializer for the contract. It sets the initial committed batches' hash and the first finalized state root, which are fundamental for the contract to function properly. Currently, there is no check on whether the function is called by the owner of the contract, which could lead to a malicious actor calling the function first.

```
function importGenesisBatch(bytes calldata _batchHeader,
    bytes32 _stateRoot) external {

    // check genesis batch header length
    require(_stateRoot ≠ bytes32(0), "zero state root");

    // check whether the genesis batch is imported
    require(finalizedStateRoots[0] == bytes32(0), "Genesis batch
    imported");

    (uint256 memPtr, bytes32 _batchHash)
    = _loadBatchHeader(_batchHeader);

    // check all fields except `dataHash` and `lastBlockHash` are zero
    unchecked {
        uint256 sum = BatchHeaderV0Codec.version(memPtr) +
            BatchHeaderV0Codec.batchIndex(memPtr) +
            BatchHeaderV0Codec.l1MessagePopped(memPtr) +
            BatchHeaderV0Codec.totalL1MessagePopped(memPtr);
        require(sum == 0, "not all fields are zero");
    }
    require(BatchHeaderV0Codec.dataHash(memPtr) ≠ bytes32(0), "zero data
    hash");
```

```
    require(BatchHeaderV0Codec.parentBatchHash(memPtr) == bytes32(0),
    "nonzero parent batch hash");

    committedBatches[0] = _batchHash;
    finalizedStateRoots[0] = _stateRoot;

    emit CommitBatch(_batchHash);
    emit FinalizeBatch(_batchHash, _stateRoot, bytes32(0));
}
```

Impact

The main implication is that the contract will not function as expected, since both the initial state root and the committed batch can be set to wrong values by the attacker. The Likelihood of this issue is set to **Low**, however, since the way the contract will theoretically be deployed should not allow for the issue to ever happen.

Recommendations

The `onlyOwner` modifier should be added to the `importGenesisBatch` function to ensure that only the owner of the contract can call it. Alternatively, any other role can be used, as long as it is ensured that the role is only assigned to a privileged address.

Remmediation

This issue has been acknowledged by Scroll Tech.

3.2 Addtional checks could be performed

- **Target**: L2StandardERC20Gateway, L2GasPriceOracle
- **Category**: Business Logic
- **Severity**: Low
- **Likelihood**: Medium
- **Impact**: Low

Description

Checks are an important part of secure smart contract development. More often than not, they form important invariants that must be maintained for the contract to function properly. Some of the contracts do not perform additional checks on variables or parameters. This could lead to unexpected behavior or even potential vulnerabilities down the development road.

In L2StandardERC20Gateway, the first deposit of a new token on a chain typically implies deploying that token. For that, the contract checks whether the `extcodesize` of the token is greater than zero (i.e., the address is a contract) and deploys the token if it is not, via the `_deployL2Token` function.

```
function finalizeDepositERC20( ... )
    external payable override onlyCallByCounterpart nonReentrant {
    bool _hasMetadata;
    (_hasMetadata, _data) = abi.decode(_data, (bool, bytes));

    bytes memory _deployData;
    bytes memory _callData;

    if (_hasMetadata) {
        (_callData, _deployData) = abi.decode(_data, (bytes, bytes));
    } else {
        require(tokenMapping[_l2Token] == _l1Token, "token mapping
    mismatch");
        _callData = _data;
    }

    if (!_l2Token.isContract()) {
        // first deposit, update mapping
        tokenMapping[_l2Token] = _l1Token;

        _deployL2Token(_deployData, _l1Token);
    }
```

```
    // ...
```

However, the contract does not check whether the `_deployData` is empty or not (as it does a few lines above via the `_hasMetadata` variable). This will lead to a revert in the `_deployL2Token` function, since it will not be able to decode the `_deployData` empty bytes array.

```
function _deployL2Token(bytes memory _deployData, address _l1Token)
    internal {
    address _l2Token =
    IScrollStandardERC20Factory(tokenFactory).deployL2Token(address(this),
    _l1Token);
    (string memory _symbol, string memory _name, uint8 _decimals)
    = abi.decode(
        _deployData,
        (string, string, uint8)
    );
```

In L2GasPriceOracle, the `setIntrinsicParams` function does not perform any checks on any of the parameters:

```
function setIntrinsicParams(
    uint64 _txGas,
    uint64 _txGasContractCreation,
    uint64 _zeroGas,
    uint64 _nonZeroGas
) public {
    require(whitelist.isSenderAllowed(msg.sender), "Not whitelisted
    sender");

    intrinsicParams = IntrinsicParams({
        txGas: _txGas,
        txGasContractCreation: _txGasContractCreation,
        zeroGas: _zeroGas,
        nonZeroGas: _nonZeroGas
    });
    // ...
}
```

Impact

The impact of this issue is low, since in both presented cases the function will either revert on its own eventually or only allow privileged users to call it. However, maintaining a consistent check pattern is important for the security of the contract as well as ensuring that the contract will not revert unexpectedly.

Recommendations

We recommend adding checks to the functions to ensure that the contract will not revert unexpectedly.

In the case of L2StandardERC20Gateway, we recommend adding a check on the `_deployData` variable to ensure that it is not empty, right before calling the `_deployL2Token` function.

```
    // ...
    if (!_l2Token.isContract()) {
        // first deposit, update mapping
        tokenMapping[_l2Token] = _l1Token;
        require(_deployData.length > 0, "deploy data is empty");
        _deployL2Token(_deployData, _l1Token);
    }
```

In the case of L2GasPriceOracle, we recommend adding a check on the parameters to ensure that they are not zero or that they are within a certain bound. For example,

```
function setIntrinsicParams(
    uint64 _txGas,
    uint64 _txGasContractCreation,
    uint64 _zeroGas,
    uint64 _nonZeroGas
) public {
    require(whitelist.isSenderAllowed(msg.sender), "Not whitelisted
    sender");

    require(_txGas > 0, "txGas is 0");
    require(_txGasContractCreation > _txGas && _txGasContractCreation >
        1e18, "txGasContractCreation is 0 or less than txGas");
    // ...
    intrinsicParams = IntrinsicParams({
```

```
        txGas: _txGas,
        txGasContractCreation: _txGasContractCreation,
        zeroGas: _zeroGas,
        nonZeroGas: _nonZeroGas
    });
    // ...
}
```

Remediation

This issue has been acknowledged by Scroll Tech and a partial fix, addressing the issue in L2GasPriceOracle, has been implemented in 1437c267.

3.3 Contracts inherit a function to renounce ownership

- **Target**: Project Wide
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Numerous contracts implement `OwnableBase`, which provides a method named `renounceOwnership` that removes the current owner (see here). This is likely not a desired feature.

Impact

If `renounceOwnership` were called, the contract would be left without an owner.

Recommendations

Override the `renounceOwnership` function:

```
function renounceOwnership() public {
    revert("This feature is not available.");
}
```

Remediation

This issue has been acknowledged by Scroll Tech.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security-related, and do not convey that we are suggesting a code change.

4.1 Differences since the initial security review

This assessment focused on the changes made to the code since the initial security review. The following is a list of those changes.

GasSwap.sol

This is a new addition to the codebase. It is a contract that allows swapping ERC-20 tokens for native gas tokens via the `swap` function:

```
function swap(PermitData memory _permit, SwapData memory _swap)
    external nonReentrant {
    require(approvedTargets[_swap.target], "target not approved");
    address _sender = _msgSender();

    // do permit
    IERC20Permit(_permit.token).permit(
        _sender,
        address(this),
        _permit.value,
        _permit.deadline,
        _permit.v,
        _permit.r,
        _permit.s
    );

    // transfer token
    IERC20(_permit.token).safeTransferFrom(_sender, address(this),
    _permit.value);

    // approve
    IERC20(_permit.token).safeApprove(_swap.target, 0);
    IERC20(_permit.token).safeApprove(_swap.target, _permit.value);
```

```
    // do swap
    uint256 _outputTokenAmount = address(this).balance;
    // solhint-disable-next-line avoid-low-level-calls
    (bool _success, bytes memory _res) = _swap.target.call(_swap.data);
    require(_success, string(concat(bytes("swap failed: "),
    bytes(getRevertMsg(_res)))));
    _outputTokenAmount = address(this).balance - _outputTokenAmount;

    // take fee
    uint256 _fee = (_outputTokenAmount * feeRatio) / PRECISION;
    _outputTokenAmount = _outputTokenAmount - _fee;
    require(_outputTokenAmount ≥ _swap.minOutput, "insufficient output
    amount");

    // tranfer ETH to sender
    (_success, ) = _sender.call{value: _outputTokenAmount}("");
    require(_success, "transfer ETH failed");

    // refund rest token
    uint256 _dust = IERC20(_permit.token).balanceOf(address(this));
    if (_dust > 0) {
        IERC20(_permit.token).safeTransfer(_sender, _dust);
    }
}
```

It makes use of `ERC20Permit`, which is a fork of the EIP-2612 proposal that adds a `permit` functionality to the ERC-20 standard. The `permit` function allows a user to approve a transfer of ERC-20 tokens without having to send an explicit transaction to do so. This is useful for gasless transactions, where the user does not have any ETH to pay for the gas fees; thus, it perfectly suits the purpose of this function.

As a general constraint to this contract, it should by default hold no ERC-20 tokens and only hold native tokens that have been taken as fees. Currently, the only way to withdraw the native tokens is to call the `withdraw` function, which can only be called by the owner of the contract.

```
function withdraw(address _token, uint256 _amount) external onlyOwner {
    if (_token == address(0)) {

        // @audit assumed this will only be used to withdraw the fees?
```

```
        (bool success, ) = msg.sender.call{value: _amount}("");
        require(success, "ETH transfer failed");
    } else {
        IERC20(_token).safeTransfer(msg.sender, _amount);
    }
}
```

L1ScrollMessenger.sol

Additions to the L1ScrollMessenger contract include the following:

1. Proof verification or the `relayMessageWithProof` function is in place:

```
{
    address _rollup = rollup;
    require(IScrollChain(_rollup).isBatchFinalized(_proof.batchIndex),
    "Batch is not finalized");
    bytes32 _messageRoot
    = IScrollChain(_rollup).withdrawRoots(_proof.batchIndex);
    require(
        WithdrawTrieVerifier.verifyMerkleProof(_messageRoot,
    _xDomainCalldataHash, _nonce, _proof.merkleProof),
        "Invalid proof"
    );
}
```

2. The `replayMessage` function has been added:

```
function replayMessage(
    address _from,
    address _to,
    uint256 _value,
    uint256 _queueIndex,
    bytes memory _message,
    uint32 _newGasLimit,
    address _refundAddress
) external payable override whenNotPaused {
```

```
    // We will use a different `queueIndex` for the replaced message.
    However, the original `queueIndex` or `nonce`
    // is encoded in the `_message`. We will check the `xDomainCalldata`
    in layer 2 to avoid duplicated execution.
    // So, only one message will succeed in layer 2. If one of the message
    is executed successfully, the other one
    // will revert with "Message was already successfully executed".
    address _messageQueue = messageQueue;
    address _counterpart = counterpart;
    bytes memory _xDomainCalldata = _encodeXDomainCalldata(_from, _to,
    _value, _queueIndex, _message);
    bytes32 _xDomainCalldataHash = keccak256(_xDomainCalldata);

    require(isL1MessageSent[_xDomainCalldataHash], "Provided message has
    not been enqueued");

    // compute and deduct the messaging fee to fee vault.
    uint256 _fee = IL1MessageQueue(_messageQueue)
    .estimateCrossDomainMessageFee(_newGasLimit);

    // charge relayer fee
    require(msg.value ≥ _fee, "Insufficient msg.value for fee");
    if (_fee > 0) {
        (bool _success, ) = feeVault.call{value: _fee}("");
        require(_success, "Failed to deduct the fee");
    }

    // enqueue the new transaction
    IL1MessageQueue(_messageQueue).appendCrossDomainMessage(_counterpart,
    _newGasLimit, _xDomainCalldata);

    // refund fee to `_refundAddress`
    unchecked {
        uint256 _refund = msg.value - _fee;
        if (_refund > 0) {
            (bool _success, ) = _refundAddress.call{value: _refund}("");
            require(_success, "Failed to refund the fee");
        }
    }
}
```

An important disclaimer is written by the Scroll Teach team, in the comments of the function, namely,

> We will use a different `queueIndex` for the replaced message. However, the original `queueIndex` or `nonce` is encoded in the `_message`. We will check the `xDomainCalldata` in layer 2 to avoid duplicated execution. So, only one message will succeed in layer 2. If one of the messages is executed successfully, the other one will revert with "Message was already successfully executed".

The duplicated execution check is handled within the L2ScrollMessenger contract in the `relayMessage` function:

```
function relayMessage(
    address _from,
    address _to,
    uint256 _value,
    uint256 _nonce,
    bytes memory _message
) external override whenNotPaused {
    // ...
    require(!isL1MessageExecuted[_xDomainCalldataHash], "Message was
    already successfully executed");
    // ...
}
```

EnforcedTxGateway.sol

This has added an enforced transaction gateway that acts as a mask for users to make actions on behalf of their L2 account on L1.

L1CustomERC20Gateway.sol

The code has been refactored such that the necessary ERC-20 token transfer on deposit was moved within the internal `_transferERC20In`, the newly added function to L1ERC20Gateway. This will be shown in the L1ERC20Gateway section below.

L1ERC20Gateway.sol

The L1ERC20Gateway contract has an additional `_transferERC20In` function, which is called by inheriting contracts (such as L1CustomERC20Gateway) to handle the ERC-20 token transfer on deposit:

```
function _transferERC20In(
    address _token,
    uint256 _amount,
    bytes memory _data
)
    internal
    returns (
        address,
        uint256,
        bytes memory
    )
{
    address _from = msg.sender;
    if (router == msg.sender) {
        // Extract real sender if this call is from L1GatewayRouter.
        (_from, _data) = abi.decode(_data, (address, bytes));
        _amount = IL1GatewayRouter(msg.sender).requestERC20(_from,
    _token, _amount);
    } else {
        // common practice to handle fee on transfer token.
        uint256 _before = IERC20(_token).balanceOf(address(this));
        IERC20(_token).safeTransferFrom(_from, address(this), _amount);
        uint256 _after = IERC20(_token).balanceOf(address(this));
        // no unchecked here, since some weird token may return arbitrary
    balance.
        _amount = _after - _before;
    }
    // ignore weird fee on transfer token
    require(_amount > 0, "deposit zero amount");

    return (_from, _amount, _data);
}
```

This is used in L1CustomERC20Gateway, L1StandardERC20Gateway, and L1WETHGateway to handle the necessary ERC-20 token transfer upon deposit (`msg.sender` -> protocol).

L1GatewayRouter.sol

This contract added the function `requestERC20`, which implements functionality to pull user allowance from the router as a gateway. It also added certain modifiers to guar-

antee that certain calls happen during the gateway call so a malicious user cannot abuse allowance given to the router.

L1StandardERC20Gateway.sol

This contract benefits from the addition of logic in two functions: `finalizeWithdrawERC20` and `_deposit`.

Upon the first withdrawal on L1 of a specific `l1Token`, the `tokenMapping` has to be initialized with a new entry pointing to the corresponding `l2Token` address, which is now done in the `finalizeWithdrawERC20` function:

```
function finalizeWithdrawERC20(
    address _l1Token,
    address _l2Token,
    address _from,
    address _to,
    uint256 _amount,
    bytes calldata _data
) external payable override onlyCallByCounterpart nonReentrant {

    require(msg.value == 0, "nonzero msg.value");
    require(_l2Token ≠ address(0), "token address cannot be 0");
    require(getL2ERC20Address(_l1Token) == _l2Token, "l2 token mismatch"
        );

    // update `tokenMapping` on first withdraw
    address _storedL2Token = tokenMapping[_l1Token];
    if (_storedL2Token == address(0)) {
        tokenMapping[_l1Token] = _l2Token;
    } else {
        require(_storedL2Token == _l2Token, "l2 token mismatch");
    }

    IERC20(_l1Token).safeTransfer(_to, _amount);

    _doCallback(_to, _data);

    emit FinalizeWithdrawERC20(_l1Token, _l2Token, _from, _to, _amount,
    _data);
```

```
}
```

Moreover, the `_data` is now forwarded to `_to` via a callback interface. This particular solution for forwarding the `_data` was recommended in the previous audit report(May 26, 2023) upon discussion with the Scroll Tech team, and it is now implemented in the `_doCallback` function:

```
function _doCallback(address _to, bytes memory _data) internal {

    if (_data.length > 0 && _to.code.length > 0) {
        IScrollGatewayCallback(_to).onScrollGatewayCallback(_data);
    }
}
```

The other addition to this contract was the refactoring and usage of the `_transferERC20In` function in the `_deposit` function, just as in the L1ERC20Gateway contract.

```
function _deposit(
    address _token,
    address _to,
    uint256 _amount,
    bytes memory _data,
    uint256 _gasLimit
) internal virtual override nonReentrant {
    require(_amount > 0, "deposit zero amount");

    // 1. Transfer token into this contract.
    address _from;
        (_from, _amount, _data) = _transferERC20In(_token, _amount, _
            data);

    // 2. Generate message passed to L2StandardERC20Gateway.
    address _l2Token = tokenMapping[_token];
    // ...
```

L1WETHGateway.sol

Similar to L1StandardERC20Gateway, the L1WETHGateway contract now uses `_transferERC20In` in the `_deposit` function as well as the `_doCallback` function within the

`finalizeWithdrawERC20` function.

L1USDCGateway.sol

Currently, no implementation of the L1USDCGateway contract exists in the codebase.

L1MessageQueue.sol

In this contract, `estimateCrossDomainMessageFee` was changed to only need `gasLimit` and an RLP encoder (in assembly) was added to compute transaction hashes (`computeTransactionHash`).

L2GasPriceOracle.sol

Some changes to L2GasPriceOracle include the addition of calculating the necessary gas fee for intrinsic cross-chain messages, now handled by the `calculateIntrinsicGasFee` function,

```
function calculateIntrinsicGasFee(bytes memory _message)
    external view override returns (uint256) {

    // @note currently we don't support contract deployment via L1
    messages.
    uint256 _txGas = uint256(intrinsicParams.txGas);
    uint256 _zeroGas = uint256(intrinsicParams.zeroGas);
    uint256 _nonZeroGas = uint256(intrinsicParams.nonZeroGas);

    uint256 gas = _txGas;
    if (_message.length > 0) {
        uint256 nz = 0;
        for (uint256 i = 0; i < _message.length; i++) {
            if (_message[i] != 0) {
                nz++;
            }
        }
        gas += nz * _nonZeroGas + (_message.length - nz) * _zeroGas;
    }
    return uint256(gas);
}
```

as well as the simplification of the `estimateCrossDomainMessageFee` function:

```
function estimateCrossDomainMessageFee(uint256 _gasLimit)
    external view override returns (uint256) {
    return _gasLimit * l2BaseFee;
}
```

The `intrinsicParams` can be set by a whitelisted `msg.sender` via the `setIntrinsicParams` function:

```
function setIntrinsicParams(
    uint64 _txGas,
    uint64 _txGasContractCreation,
    uint64 _zeroGas,
    uint64 _nonZeroGas
) public {
    require(whitelist.isSenderAllowed(msg.sender), "Not whitelisted
    sender");

    intrinsicParams = IntrinsicParams({
        txGas: _txGas,
        txGasContractCreation: _txGasContractCreation,
        zeroGas: _zeroGas,
        nonZeroGas: _nonZeroGas
    });

    emit IntrinsicParamsUpdated(_txGas, _txGasContractCreation, _zeroGas,
    _nonZeroGas);
}
```

MultipleVersionRollupVerifier.sol

This is an interface that is still in progress to handle and verify multiple rollup proofs using different verifiers.

ScrollChain.sol

The ScrollChain contract has been heavily modified since the initial security review. The following is a list of those changes:

- The entire layout of the `commitBatch` function has been changed. It now uses a BatchHeaderV0Codec library for handling storing of the variables in memory, rather than using a `storage` variable. To summarize, the parent batch header is

copied from calldata to memory, starting at a predetermined `batchPtr` pointer. Then, `_chunksLength` number of hashes (i.e., of chunks) are stored starting from the `dataPtr` pointer. Memory is reserved from `dataPtr` to `dataPtr + _chunkLength * 32` for the chunk hashes. Finally, starting at `newBatchptr` pointer, the new batch header is used to store the new header, which is then used to compute the batch hash, stored at the `commitedBatches[_batchIndex]` position in storage.

- The `revertBatch` function now checks whether the `index` of the `batch` that is to be removed is greater than the `lastFinalizedBatchIndex`. Since it is assumed that the indexes are always increasing, this check ensures that the `batch` is not already finalized.
- The `finalizeBatchWithProof` function now handles popping finalized messages from the L1MessageQueue, such that it keeps the L1MessageQueue in sync with ScrollChain. This is done by calling the `popCrossDomainMessage` function in L1MessageQueue.
- Some additional `setters` were added for the various state variables and mappings that the contract has(e.g., `isProver`, `verifier`, `maxNumL2TxInChunk`, etc.)
- Messages are now stored in `chunks`. These chunks are then individually committed via the `commitBatch`; thus a batch now essentially consists of multiple chunks. This modification is mainly implemented via the `_commitChunk` function, which handles parsing the `chunk` from calldata and storing it in memory temporarily.

L2ScrollMessenger.sol

In L2ScrollMessenger, several layout slots have been changed to account for differences in the structure of the contract. Also, it removed the refund functionality to tx.origin and deleted the custom non-reentrant modifier and opted to use the standard one.

L2StandardERC20Gateway.sol

This has fixed a vulnerability in the handling of token data in `tokenMappings` when failed calls happen.

L2USDCGateway.sol

This contract is a new addition to the codebase. It allows the deposit and withdrawal of USDC tokens on L2 by inheriting from the L2ERC20Gateway contract and overriding `finalizeDepositERC20` and `withdraw` with additional checks such that only USDC is allowed to be used as `l1Token` or `l2Token`.

BatchHeaderV0Codec.sol

This is a library added to handle the batches and chunks; this allows for neater programming and getters for the batches. It has added a function that loads and validates a `batch` based on the size of the `skipBitmap`, relative to the number of `l1MessagesPopped`, and added helpers that access several offsets into the batch and compute hashes.

ChunkCodec.sol

This has added a function that validates chunks, and ensures that the number of bytes and number of chunks correspond, and has added helpers that access several variables in the chunks and compute hashes.

ScrollGatewayBase.sol

The ScrollGatewayBase benefits from the `_doCallback` hook, which now handles the data forwarding to the destination contracts on each of the chains.

```
function _doCallback(address _to, bytes memory _data) internal {

    if (_data.length > 0 && _to.code.length > 0) {
        IScrollGatewayCallback(_to).onScrollGatewayCallback(_data);
    }
}
```

This modification came as a result of the previous audit report(May 26, 2023), where it was recommended to implement a callback interface for the Scroll contracts, such that the `data` can be forwarded to the destination contracts on each of the chains, rather than performing a `call` with arbitrary `data` on an arbitrary address.

WithdrawTrieVerifier.sol

This is a simple implementation that verifies that a Merkle tree is correct.

4.2 The RLP fuzzer

A fuzzer was written in Forge Foundry that tested the implementation `computeTransactionHash` in L1MessageQueue, specifically the transaction encoder that encodes `0x7e` followed by the RLP-encoded transaction values. The Forge project generates random data to be encoded and verifies that `computeTransactionHash` matches the output of the `pyrlp` Python implementation.

After many iterations, we determined that it is extremely unlikely for there to be security issues in the RLP encoder.

4.3 The `patriciaMerkleProof` fuzzer

A fuzzer was created that takes proofs from popular addresses on the blockchain and verifies that the `patriciaMerkleProof` verifier generates the correct expected value and state root.

Around 80,000 proofs were fuzzed/tested and resulted in no errors.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

Moreover, the threat model has been written for the newly added contracts over the previous audit. The threat model for the updated/unchanged contracts can be found in the previous audit report(May 26, 2023).

5.1 Module: BatchHeaderV0Codec.sol

Function: `loadAndValidate(byte[] _batchHeader)`

This is a function to load and validate the batch header from calldata to memory.

Inputs

- `_batchHeader`
  - **Control**: Controlled by calling the function.
  - **Constraints**: Length must be >= 89.
  - **Impact**: The batch header is copied to memory.

Branches and code coverage (including function calls)

Intended branches

- Copy from calldata to memory via `calldatacopy` using the `offset` field of `_batchHeader` to determine the start of the batch header in calldata as well as the `length` field of `_batchHeader` to determine the length of the batch header.
  ☑ Test coverage
- Store the batch header length in `length`.
  ☑ Test coverage
- Should check the length of the bitmap.
  ☑ Test coverage

Negative behavior

- Should not allow the length of the batch header to mismatch the calculated bitmap length.
  ☑ Negative test

5.2 Module: EnforcedTxGateway.sol

Function: `initialize(address _queue, address _feeVault)`

This allows initializing the contract.

Inputs

- `_queue`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the `messageQueue` contract.
- `_feeVault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the fee vault contract.

Branches and code coverage (including function calls)

Intended branches

- Call all necessary underlying initializer functions.
  ☑ Test coverage
- Set the `messageQueue` variable.
  ☑ Test coverage
- Set the `feeVault` variable.
  ☑ Test coverage

Negative behavior

- Should not allow calling multiple times.
  ☑ Negative test

Function: `sendTransaction(address _sender, address _target, uint256 _value, uint256 _gasLimit, byte[] _data, byte[] _signature, address _refundAddress)`

This allows sending an enforced transaction to L2 with a signature.

Inputs

- `_sender`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked against the recovered signer.
  - **Impact**: The address of the sender who will initiate this transaction in L2.
- `_target`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address of the target contract to call in L2.
- `_value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The value passed.
- `_gasLimit`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The maximum gas should be used for this transaction in L2.
- `_data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The calldata passed to the target contract.
- `_signature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the recovered signer matches the `_sender`.
  - **Impact**: The signature for the transaction.
- `_refundAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address to refund the exceeded fee.

Branches and code coverage (including function calls)

Intended branches

- Call `_sendTransaction`, which handles the actual sending of the transaction.
  ☑ Test coverage
- Assure that `_sender` is not `address(0)`. This is enforced by `recover`.
  ☐ Test coverage

Negative behavior

- Should not allow passing an invalid signature. This is enforced by checking that the recovered signer matches the `_sender`.
  ☑ Negative test

Function: `sendTransaction(address _target, uint256 _value, uint256 _gasLimit, byte[] _data)`

This allows sending an enforced transaction to L2.

Inputs

- `_target`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The address of the target contract to call in L2.
- `_value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The value passed.
- `_gasLimit`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The maximum gas should be used for this transaction in L2.
- `_data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The calldata passed to the target contract.

Branches and code coverage (including function calls)

Intended branches

- Call `_sendTransaction`, which handles the actual sending of the transaction.
  ☑ Test coverage

Function: `setPaused(bool _status)`

This allows pausing or unpausing the contract.

Inputs

- `_status`

- **Control**: Fully controlled by the owner.
- **Constraints**: None.
- **Impact**: Pauses this contract if it is true; otherwise it unpauses this contract.

Branches and code coverage (including function calls)

Intended branches

- Call `_pause` if `_status` is true.
  ☑ Test coverage
- Call `_unpause` if `_status` is false.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test

Function: `updateFeeVault(address _newFeeVault)`

This allows updating the address of the fee vault.

Inputs

- `_newFeeVault`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The address of the new fee vault contract.

Branches and code coverage (including function calls)

Intended branches

- Set the `feeVault` variable.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test

Function: `_sendTransaction(address _sender, address _target, uint256 _value, uint256 _gasLimit, byte[] _data, address _refundAddress)`

The internal function responsible for actually sending the transaction.

Inputs

- `_sender`
  - **Control**: Controlled by the caller.
  - **Constraints**: Depends on the calling function. No checks here.
  - **Impact**: The sender of the transaction in L2.
- `_target`
  - **Control**: Controlled by the caller.
  - **Constraints**: Depends on the calling function. No checks here.
  - **Impact**: The address of the target contract to call in L2.
- `_value`
  - **Control**: Controlled by the caller.
  - **Constraints**: Depends on the calling function. No checks here.
  - **Impact**: The value passed in L2.
- `_gasLimit`
  - **Control**: Controlled by the caller.
  - **Constraints**: Depends on the calling function. No checks here.
  - **Impact**: The maximum gas should be used for this transaction in L2.
- `_data`
  - **Control**: Controlled by the caller.
  - **Constraints**: Depends on the calling function. No checks here.
  - **Impact**: The calldata passed to the target contract.
- `_refundAddress`
  - **Control**: Controlled by the caller.
  - **Constraints**: Depends on the calling function. No checks here.
  - **Impact**: The address to refund the exceeded fee.

Branches and code coverage (including function calls)

Intended branches

- Assumes that `_value` is not greater than `msg.value`. This is not enforced here; however, it should be enforced in the off-chain component.
  ☐ Test coverage
- Call `IL1MessageQueue.estimateCrossDomainMessageFee` to get the fee.
  ☑ Test coverage
- Check that `msg.value` is greater than or equal to the fee.
  ☑ Test coverage

Negative behavior

- Should not allow sending a transaction with an insufficient fee. This is enforced by checking that `msg.value` is greater than or equal to the fee.
  ☑ Negative test
- Should not allow sending a message on behalf of someone else. This is theoretically enforced in calling functions.
  ☑ Negative test
- Should revert if refund fails. This is enforced by using `unchecked`.
  ☑ Negative test

5.3 Module: GasSwap.sol

Function: `swap(PermitData _permit, SwapData _swap)`

This allows swapping ERC-20 tokens for native ETH.

Inputs

- `_permit`
  - **Control**: Fully controlled by the user.
  - **Constraints**: Assumed to be valid — also checked by the `ERC20Permit` contract for validity.
  - **Impact**: The permit that the contract will use to spend the user's tokens.
- `_swap`
  - **Control**: Fully controlled by the user.
  - **Constraints**: Checked that the target is approved by the contract owner.
  - **Impact**: The swap data that the contract will use to perform the swap.

Branches and code coverage (including function calls)

Intended branches

- Allow anyone to perform a swap via a specified target contract.
  ☑ Test coverage
- Send the swapped ETH back to the user.
  ☑ Test coverage
- Deplete the user's balance of the token they are swapping.
  ☑ Test coverage
- Deplete the user's allowance of the token they are swapping.
  ☑ Test coverage
- Transfer any remaining tokens back to the user.
  ☑ Test coverage

Negative behavior

- Should not allow a swap to be performed if the target is not approved by the contract owner.
  ☑ Negative test
- Should not allow a swap to be performed if the user does not have enough tokens to swap.
  ☑ Negative test
- Should not allow a swap to be performed if the user does not have enough allowance to swap.
  ☑ Negative test
- Should not allow a swap to be performed if the user does not have enough ETH to pay the fee.
  ☑ Negative test

Function: `updateApprovedTarget(address _target, bool _status)`

This updates the status of a target address.

Inputs

- `_target`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The target address to update.
- `_status`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The new status of the target address.

Branches and code coverage (including function calls)

Intended branches

- Update the status of the target address.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the owner to call this function.
  ☑ Negative test

Function: `updateFeeRatio(uint256 _feeRatio)`

Allows owner to update the fee ratio.

Inputs

- `_feeRatio`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The new fee ratio.

Branches and code coverage (including function calls)

Intended branches

- Update the `feeRatio` variable to the new fee ratio.
  ☑ Test coverage

Negative behaviour

- Should not allow anyone other than the owner to update the fee ratio.
  ☑ Negative test

Function: `withdraw(address _token, uint256 _amount)`

This allows the owner to withdraw tokens from the contract.

Inputs

- `_token`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The token that the owner wants to withdraw.
- `_amount`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The amount of tokens that the owner wants to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Should be able to withdraw ETH.
  ☑ Test coverage

- Should be able to withdraw tokens.
  ☑ Test coverage
- Should be used to withdraw either fees or "stuck" tokens. Assumed no malicious intent on behalf of the owner.
  ☑ Test coverage
- Increase the owner's balance of the token they are withdrawing by amount.
  ☑ Test coverage
- Decrease the contract's balance of the token they are withdrawing by amount.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test

5.4 Module: L2USDCGateway.sol

Function: `finalizeDepositERC20(address _l1Token, address _l2Token, address _from, address _to, uint256 _amount, byte[] _data)`

This allows the finalization of a deposit from L1 to L2.

Inputs

- `_l1Token`
  - **Control**: Full control.
  - **Constraints**: `l1Token == l1USDC`.
  - **Impact**: L1USDC.
- `_l2Token`
  - **Control**: Full control.
  - **Constraints**: `l2Token == l2USDC`.
  - **Impact**: L2USDC.
- `_from`
  - **Control**: Full control.
  - **Constraints**: N/A (checked in L2).
  - **Impact**: Sender from L1.
- `_to`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Destination address.

- `_amount`
  - **Control**: Full control.
  - **Constraints**: Checked that it is `msg.value`.
  - **Impact**: Amount to be deposited.
- `_data`
  - **Control**: Full control.
  - **Constraints**: N/A (checked in L2).
  - **Impact**: Data to be passed on to the recipient.

Branches and code coverage (including function calls)

Intended branches

- `msg.value` should be equal to `_amount`.
  ☑ Test coverage
- `_l1Token` should be equal to `l1USDC`.
  ☑ Test coverage
- `_l2Token` should be equal to `l2USDC`.
  ☑ Test coverage
- `_amount` should be greater than zero.
  ☑ Test coverage
- Should forward `_data` to `_to`.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the `counterpart`.
  ☑ Negative test
- Should not be callable multiple times for the same deposit action in L1.
  ☑ Negative test
- Should not be callable with a zero `_amount`.
  ☑ Negative test

Function: `pauseDeposit(bool _paused)`

This allows the owner to pause deposits to the contract.

Inputs

- `_paused`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.

- **Impact**: The new status `true` means paused, and `false` means not paused.

Branches and code coverage (including function calls)

Intended branches

- Set the `depositPaused` variable to `_paused`.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the owner to call this function.
  ☑ Negative test

Function: `pauseWithdraw(bool _paused)`

This allows owner to pause withdraws from the contract.

Inputs

- `_paused`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: The new status `true` means paused, and `false` means not paused.

Branches and code coverage (including function calls)

Intended branches

- Set the `withdrawPaused` variable to `_paused`.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the owner to call this function.
  ☑ Negative test

Function: `_withdraw(address _token, address _to, uint256 _amount, byte[] _data, uint256 _gasLimit)`

This facilitates the withdraw and transfer of USDC from Layer 1 to Layer 2.

Inputs

- `_token`
  - **Control**: Full control.

  - **Constraints**: Checked it is the USDC address.
  - **Impact**: The ERC-20 token to be withdrawn.
- `_to`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The destination address for the withdraw.
- `_amount`
  - **Control**: Full control.
  - **Constraints**: `_amount > 0`.
  - **Impact**: The amount of the ERC-20 token to be withdrawn.
- `_data`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The data to be passed to the recipient.
- `_gasLimit`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The gas limit for the withdraw.

Branches and code coverage (including function calls)

Intended branches

- Should decrease the balance of USDC by `_amount` for the `msg.sender`.
  ☑ Test coverage
- Should increase the balance of `native` by `_amount` for the `messenger`. This is because `msg.value` will be used as the `fee` for the `messenger`, whereas the `_amount` will be used as the actual amount to be withdrawn.
  ☑ Test coverage
- Should forward the payload on to the messenger.
  ☑ Test coverage

Negative behavior

- Should not permit withdrawing tokens other than USDC.
  ☑ Test coverage

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Scroll contracts, we discovered three findings. No critical issues were found. One was of medium impact, one was of low impact, and the remaining finding was informational in nature. Scroll Tech acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.