Pyth Express Relay

Security Assessment

June 28th, 2024 — Prepared by OtterSec

Robert Chen r@osec.io

Nicholas R. Putra nicholas@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Pyth Foundation engaged OtterSec to assess the `express-relay` and `opportunity-adapter` programs. This assessment was conducted between June 1st and June 21st, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 8 findings throughout this audit engagement.

In particular, we identified a critical vulnerability, where the execution of an opportunity allows arbitrary calls to any contract, potentially resulting in loss of approved funds (OS-PER-ADV-00), and a high-risk issue regarding the possibility of performing a gas griefing attack via the multi-call feature (OS-PER-ADV-01). Furthermore, we highlighted the utilization of the transfer function which limits the functionality as it may not be future-proof, due to its fixed gas consumption restriction, resulting in reverts if the transaction surpasses the gas limit (OS-PER-ADV-02).

We also recommended optimizing the overall signature validation process (OS-PER-SUG-03) and suggested removing a redundant overflow check (OS-PER-SUG-01). Additionally, we advised the inclusion of a withdraw function to enable the owner to transfer `ETH` from the balance of the contract (OS-PER-SUG-02).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/pyth-network/per. This audit was performed against commit 6d982c4.

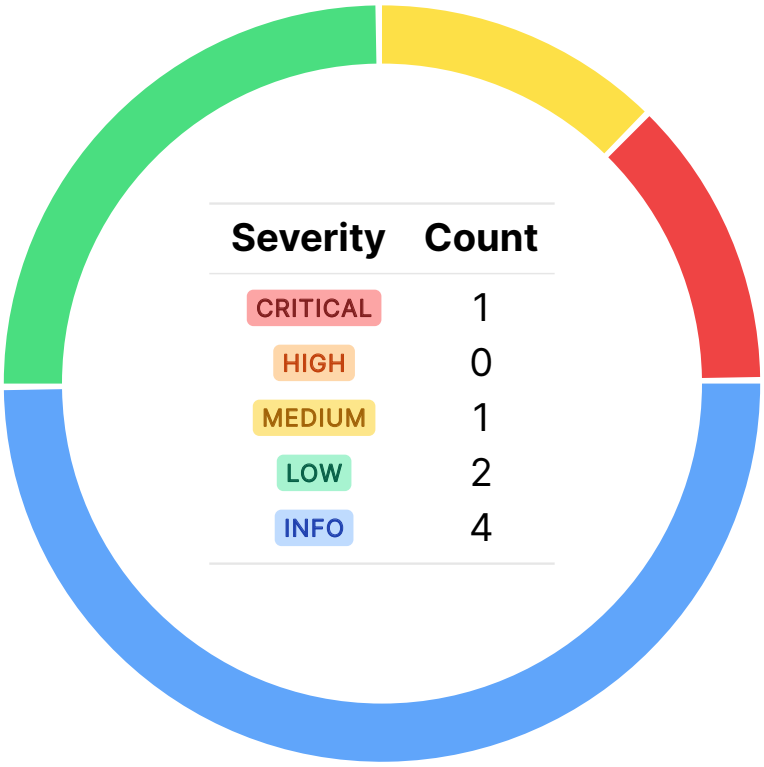
A brief description of the programs is as follows:

Name	Description
express-relay	A program to facilitate off-chain auctions of permissioned protocol actions.
opportunity-adapter	A program to allow searchers to directly interact with on-chain programs without deploying their own contracts. Serves as an ”adapter” to opportunities.

03 — Findings

Overall, we reported 8 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	0
MEDIUM	1
LOW	2
INFO	4

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-PER-ADV-00	CRITICAL	RESOLVED	`OpportunityAdapter` allows arbitrary calls to any contract via `_callTargetContract` potentially resulting in loss of allowance funds.
OS-PER-ADV-01	MEDIUM	RESOLVED	`multicall` is vulnerable to gas griefing attacks, where an attacker may submit `multicall` with a single call returning a massive amount of data, which will consume a large portion of the gas budget.
OS-PER-ADV-02	LOW	RESOLVED	The utilization of `transfer` limits the functionality as it may not be future-proof, due to its fixed gas consumption restriction, resulting in random reverts if the transaction surpasses the gas limit.
OS-PER-ADV-03	LOW	RESOLVED	Both `doLiquidate` in `SearcherVault`, and `executeOpportunity` in `OpportunityAdapter` lack a `nonReentrant` modifier.

Unsafe Arbitrary Call CRITICAL OS-PER-ADV-00

Description

`executeOpportunity` in `OpportunityAdapter` poses a significant security risk due to its use of `_callTargetContract`. This function enables arbitrary calls to external contracts, which introduces considerable security vulnerabilities, particularly in the misuse of `ERC20` allowances. `_callTargetContract` allows the contract to perform arbitrary calls to an external contract specified by the `targetContract` address with the provided `targetCalldata` and `targetCallValue`. It executes any `calldata` on the specified contract. While this functionality offers flexibility, it is prone to misuse.

>_ OpportunityAdapter.sol solidity

```solidity
function _callTargetContract(
    address targetContract,
    bytes calldata targetCalldata,
    uint256 targetCallValue
) internal {
    (bool success, bytes memory returnData) = targetContract.call{
        value: targetCallValue
    }(targetCalldata);
    if (!success) {
        revert TargetCallFailed(returnData);
    }
}
```

If `params.targetContract` points to an `ERC20` token contract and `params.targetCalldata` is crafted to call `transferFrom` on this contract, it may exploit the allowances of other users. By crafting the `ExecutionParams` to cause `_callTargetContract` to execute a call to the target token contract, a malicious user may transfer a previously approved user's allowance into `OpportunityAdapter`, draining the allowance funds of other users.

Proof of Concept

1. User `A` approves `OpportunityAdapter` to spend their tokens (`USDC`).
2. The attacker crafts `ExecutionParams` such that `params.targetContract` is the address of the `USDC` contract, and `params.targetCalldata` encodes `transferFrom` to transfer tokens from `A` to `OpportunityAdapter`.
3. Consequently, when the opportuniy is executed, `_callTargetContract` invokes `transferFrom` , transferring tokens from user `A` to `OpportunityAdapter`.

4. Thus, these tokens may be treated as the buy tokens and as a result `OpportunityAdapter` will transfer these tokens to the attacker within `_validateAndTransferBuyTokens`, effectively utilizing user `A` allowance to get additional tokens.

Remediation

Utilize the factory pattern to deploy new instances of `OpportunityAdapter` for each user via the `Create2` opcode, which allows for the deployment of contracts with deterministic addresses. By deploying a new `OpportunityAdapter` for each user, each contract operates in isolation. This ensures that allowances set by one user will not be exploited by another, as each instance of `OpportunityAdapter` will only interact with the tokens and allowances of the user who controls it.

Patch

Resolved in #94.

Gas Griefing Via Multicall MEDIUM OS-PER-ADV-01

Description

There is a potential gas griefing vulnerability in `multicall` within `ExpressRelay`. When processing `multicall`, the returned data after each call to `callWithBid` is stored in memory within `result`. A large `result` size significantly increases the gas cost of the entire `multicall` transaction. This issue is particularly concerning since `multicall` does not limit the size of `result` that stores the data returned by external calls.

>_ ExpressRelay.sol solidity

```
function multicall(
    bytes calldata permissionKey,
    MulticallData[] calldata multicallData
)[...]
{
    [...]
    for (uint256 i = 0; i < multicallData.length; i++) {
        try
            // callWithBid will revert if call to external contract fails or if bid conditions
                ↪  not met
            this.callWithBid(multicallData[i])
        returns (bool success, bytes memory result) {
            multicallStatuses[i].externalSuccess = success;
            multicallStatuses[i].externalResult = result;
        } catch Error(string memory reason) {
            multicallStatuses[i].multicallRevertReason = reason;
        }
        [...]
    }
    [...]
}
```

A malicious actor may exploit this by including a single call in `multicall` that returns an excessively large `result`. Processing this large `result` will consume a significant portion of the transaction's gas budget. If the gas cost of processing the large `result` exceeds the available gas budget, the entire `multicall` transaction may fail, disrupting the execution.

Remediation

Implement the `ExcessivelySafeCall` library and invoke `excessivelySafeCall` instead of within `callWithBid`. This will restrict the amount of data that may be copied from the external call's return value.

Patch

Fixed in f0a9a7b.

Utilization Of Depreciated Function Call LOW OS-PER-ADV-02

Description

Currently, the code base invokes `transfer` for sending funds to contracts as shown below. `transfer` is bound by a gas stipend limitation (around 2300 gas) that restricts what the receiving contract may do before the transaction fails. Additionally, the gas stipend for transfer may change in future Ethereum upgrades. Relying on it for reentrancy protection may become unreliable. Rather, implementing reentrancy protection manually along with `call` ensures the system remains secure regardless of gas stipend changes.

ExpressRelay.sol solidity

```
function multicall(
    bytes calldata permissionKey,
    MulticallData[] calldata multicallData
)[...]
{
    [...]
    // pay the relayer
    uint256 feeRelayer = ((totalBid - feeProtocol) *
        state.feeSplitRelayer) / state.feeSplitPrecision;
    if (feeRelayer > 0) {
        payable(state.relayer).transfer(feeRelayer);
    }
}
```

Moreover, `call` provides greater control over interactions with the receiving contract. It allows for more precise definition of the gas limit and handling of potential return values, enabling more complex interactions.

Remediation

Replace `transfer` with `call`.

Patch

Fixed in c8bce5b.

Re-entrancy Attack LOW

OS-PER-ADV-03

Description

`doLiquidate` in `SearcherVault` and `executeOpportunity` in `OpportunityAdapter` may be susceptible to reentrancy. `doLiquidate` and `OpportunityAdapter` utilize external calls to perform liquidation and execute an opportunity, respectively. Since they call `_useSignature()` after performing the external calls, it may be possible to re-enter the functions and repeatedly perform liquidation of the same vault or attempt to execute the same opportunity. This may drain the collateral by utilizing the same signature, as it is not marked as used yet, enabling it to pass the signature verification.

Remediation

Add the `nonReentrant` modifier to both `doLiquidate` and `OpportunityAdapter`.

Patch

Fixed in 154ba36 and 4a1f623.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-PER-SUG-00	`approve` may fail with some `ERC20` tokens if they require a zero allowance before setting a new value, resulting in unexpected reverts.
OS-PER-SUG-01	There is an unnecessary overflow check within `callWithBid` that should be removed due to Solidity's built-in overflow checks.
OS-PER-SUG-02	`ExpressRelay` lacks a withdraw function to enable the owner to transfer `ETH` from the balance of the contract.
OS-PER-SUG-03	Recommendations to optimize the overall signature validation process in `SearcherVault` and `OpportunityAdapter`.

Incompatibility Risk With Approve Call OS-PER-SUG-00

Description

`SearcherVault` and `OpportunityAdapter` will benefit from using `forceApprove` instead of `approve` for improved safety with specific `ERC20` tokens. The current implementation utilizes `approve` which assumes standard `ERC20` token behavior. However, some `ERC20` tokens have non-standard behavior regarding allowance setting. Specifically, some tokens require the allowance to be set to zero before setting a new non-zero allowance. Thus, if the contract previously interacted with the same token and has a non-zero allowance set, the `approve` call may revert.

>_ SigVerify.sol solidity

```solidity
// Use of approve in SigVerify Contract
function doLiquidate(
    [...]
) public payable {
    [...]
    address payable vaultContract = payable(tokenVault);
    Vault memory vault = TokenVault(vaultContract).getVault(vaultId);
    address tokenDebt = vault.tokenDebt;
    uint256 tokenAmount = vault.amountDebt;
    IERC20(tokenDebt).approve(vaultContract, tokenAmount);
    [...]
}
```

Even though approval is set back to zero especially every transaction of `OpportunityAdapter`, the issue mentioned in ADV-00 may be exploited to create approval of a non-zero amount, in which case the current implementation of `approve` will fail, if it encounters `ERC20` tokens with this non-standard behavior.

Remediation

Utilize `forceApprove` which attempts to set the allowance directly to the desired value. If the initial attempt fails due to non-standard token behavior, it attempts to reset the allowance to zero first and then retries the desired approval.

Patch

Fixed in cc5e5de.

Removal Of Redundant Code OS-PER-SUG-01

Description

`ExpressRelay::callWithBid` checks if the final balance is greater than the initial balance. While the check (`balanceFinalEth >= balanceInitEth`) may seem intuitive, it is redundant due to Solidity's built-in overflow checks introduced in versions 0.8 onwards. Thus, Solidity's built-in overflow checks provide a robust safety mechanism, rendering the additional check redundant.

>_ *ExpressRelay.sol* solidity

```solidity
function callWithBid(
    MulticallData calldata multicallData
) public payable returns (bool, bytes memory) {
    [...]
    if (success) {
        uint256 balanceFinalEth = address(this).balance;
        // ensure that this contract was paid at least bid ETH
        require(
            (balanceFinalEth - balanceInitEth >= multicallData.bidAmount) &&
                (balanceFinalEth >= balanceInitEth),
            "invalid bid"
        );
    }
    return (success, result);
}
```

Remediation

Remove the check in `ExpressRelay::callWithBid`.

Patch

Fixed in df58fb7.

Missing Owner Withdraw Functionality OS-PER-SUG-02

Description

The current implementation of `ExpressRelay` does not offer a direct way for the owner to claim accumulated `ETH`. Thus, while the contract receives `ETH` through `receive` and `callWithBid` validation process, there is no way for the owner to directly retrieve these funds. This may result in a situation where `ETH` gets locked in the contract indefinitely.

>_ ExpressRelay.sol solidity

```solidity
function callWithBid(
    MulticallData calldata multicallData
) public payable returns (bool, bytes memory) {
    [...]
    uint256 balanceInitEth = address(this).balance;
    (bool success, bytes memory result) = multicallData.targetContract.call(
        multicallData.targetCalldata
    );
    [...]
    return (success, result);
}

receive() external payable {
    emit ReceivedETH(msg.sender, msg.value);
}
```

Remediation

Add a dedicated withdrawal function with proper access control.

Patch

Resolved in #98.

Optimization Of Signature Verification OS-PER-SUG-03

Description

1. While `ECDSA.recover` itself does not inherently allow for malleability, relying solely on it within `SearcherVault` and `OpportunityAdapter` for signature verification introduces a weakness. Include a unique `nonce` within the signed message to significantly improve the security of the signature verification process, preventing replay attacks.
2. As a potential optimization, within `SearcherVault` and `OpportunityAdapter`, call `_useSignature` only after signature validation and only if the signature is from `ExpressRelay` to save on gas.
3. Currently, `_useSignature` directly marks a signature as used without first checking if it was unused. Modify `_useSignature` to check if a signature is already marked as used.

>_ SigVerify.sol solidity

```solidity
function _useSignature(bytes memory signature) internal {
    _signatureUsed[signature] = true;
}
```

Remediation

Implement the above-mentioned modifications.

Patch

Resolved by using Permit2 in #94.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.