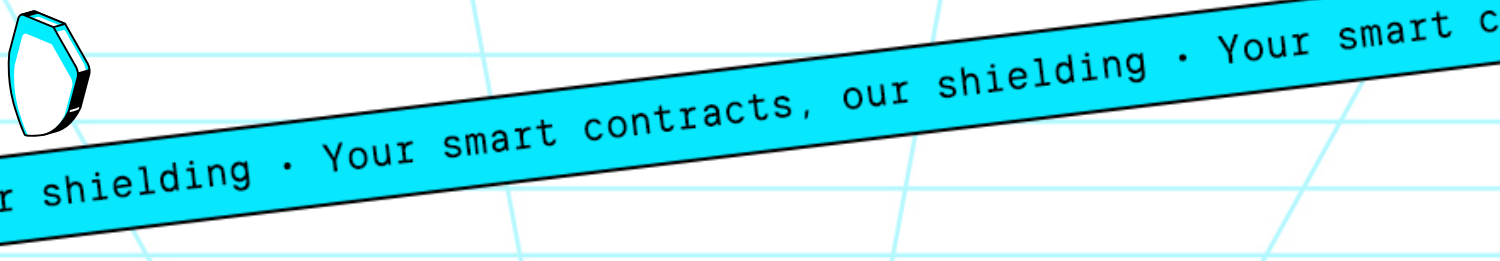
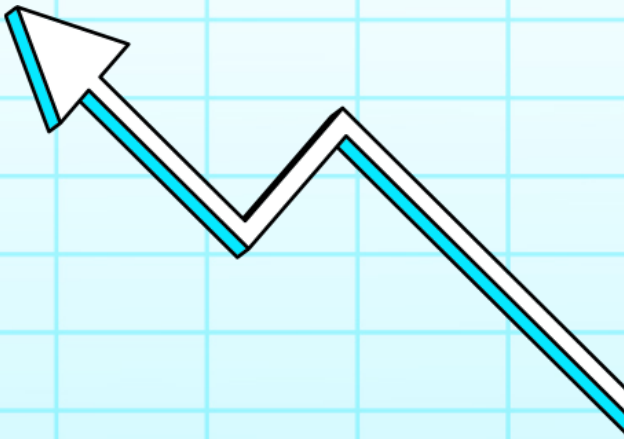


shieldify



Shiny
Payment Router

SECURITY REVIEW

Date: 22 January 2026

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Shiny - Payment Router

Shiny is an all-in-one experience for collectors, making luxury assets as easy to access as digital ones through verifiable mystery packs. Launching with Pokémon, the world's most valuable IP, as its wedge into a broader collectibles market. With first-party supply, guaranteed liquidity, and a founder-led network, the company is built to scale fast and capture the future of collecting, all on shiny.com.

Learn more about Shiny's concept and the technicalities behind it: here

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 1 days with a total of 24 hours dedicated to the audit by the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying one Medium and one Info severity finding. They're related to batch payment handling issues, specifically around Order ID reuse and flawed safety cap checks that could break batch execution.

5.1 Protocol Summary

Project Name	**Shiny - Payment Router**
Repository	N/A
Type of Project	RWA, Treasury, EIP-712
Security Review Timeline	1 day

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
PaymentRouter.sol	136
Total	136

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Medium** issues: **1**
- **Info** issues: **1**

ID	Title	Severity	Status
[M-01]	Same Order ID Can Be Used Twice in Batch Payments	Medium	Fixed
[I-01]	Batch Safety Cap Check Is Redundant and Can Overflow, Bricking Batch Payments	Info	Fixed

7. Findings

[M-01] Same Order ID Can Be Used Twice in Batch Payments

Severity

Medium Risk

Description

The `payOrdersBatch()` function validates all orders against the initial state before updating any state in a subsequent loop. This allows duplicate `orderIds` in a single batch to bypass the `OrderAlreadyProcessed` check.

Location of Affected Code

File: src/PaymentRouter.sol

```
function payOrdersBatch() {
    // code
    for (uint256 i = 0; i < len; i++) {
        _validateOrderLogic(orderIds[i], amounts[i]);
        totalAmount += amounts[i];
    }
    // code
}
```

Impact

Duplicate processing of orders leads to double payments and duplicate event emissions, potentially triggering double fulfilment by off-chain systems.

Proof of Concept

Create a new file under `test/PaymentRouterReproduce.t.sol` and write the exploit.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "../src/PaymentRouter.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

// Simple Mock USDC for testing
contract MockUSDC is ERC20 {
    constructor() ERC20("USD Coin", "USDC") {}
    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }
}

contract PaymentRouterReproduce is Test {
    PaymentRouter public router;
    MockUSDC public usdc;
```

```
uint256 internal backendPk;
address internal backendSigner;

uint256 internal userPk;
address internal user;

address internal treasury;

bytes32 private constant BATCH_PAYMENT_TYPEHASH = keccak256("
   BatchPayment(bytes32 batchHash,uint256 totalAmount,uint256
   validUntil,address user,address treasury)");

uint256 MAX_PAYMENT = 1000 * 1e6;

function setUp() public {
    backendPk = 0xB0B;
    backendSigner = vm.addr(backendPk);

    userPk = 0xCAFE;
    user = vm.addr(userPk);

    treasury = makeAddr("treasury");

    usdc = new MockUSDC();
    router = new PaymentRouter(
        address(usdc),
        treasury,
        backendSigner,
        MAX_PAYMENT
    );

    usdc.mint(user, 10000 * 1e6);

    vm.prank(user);
    usdc.approve(address(router), type(uint256).max);
}

function _signBatch(
    uint256 pk,
    uint256[] memory orderIds,
    uint256[] memory amounts,
    uint256 validUntil,
    address _user,
    address _treasury
) internal view returns (bytes memory) {
    uint256 total = 0;
    for(uint256 i=0; i<amounts.length; i++) total += amounts[i];
```

```
        bytes32 batchHash = keccak256(abi.encode(orderIds, amounts));

        bytes32 structHash = keccak256(abi.encode(
            BATCH_PAYMENT_TYPEHASH,
            batchHash,
            total,
            validUntil,
            _user,
            _treasury
        ));

        bytes32 digest = _getDigest(structHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(pk, digest);
        return abi.encodePacked(r, s, v);
    }

    function _getDigest(bytes32 structHash) internal view returns (
       bytes32) {
        bytes32 EIP712_DOMAIN_TYPEHASH = keccak256("EIP712Domain(string
           name,string version,uint256 chainId,address verifyingContract)
           ");
        bytes32 nameHash = keccak256("PaymentRouter");
        bytes32 versionHash = keccak256("1");
        uint256 chainId = block.chainid;
        address verifyingContract = address(router);

        bytes32 domainSeparator = keccak256(abi.encode(
            EIP712_DOMAIN_TYPEHASH,
            nameHash,
            versionHash,
            chainId,
            verifyingContract
        ));

        return keccak256(abi.encodePacked(
            "\x19\x01",
            domainSeparator,
            structHash
        ));
    }

    function test_Exploit_DuplicateOrderIdsInBatch() public {
        uint256[] memory ids = new uint256[](2);
        ids[0] = 777;
        ids[1] = 777; // Duplicate ID!
```

```
        uint256[] memory amounts = new uint256[](2);
        amounts[0] = 10 * 1e6;
        amounts[1] = 10 * 1e6;

        uint256 validUntil = block.timestamp + 1 hours;

        // Backend signs the batch (assuming backend checks are also
           loose or bypassed,
        // OR simply demonstrating that the contract doesn't enforce
           uniqueness)
        bytes memory sig = _signBatch(backendPk, ids, amounts, validUntil
           , user, treasury);

        uint256 initialTreasuryBalance = usdc.balanceOf(treasury);

        vm.prank(user);
        router.payOrdersBatch(ids, amounts, validUntil, sig);

        // Verification:
        // Treasury should have received 20 USDC
        assertEq(usdc.balanceOf(treasury), initialTreasuryBalance + 20 *
           1e6);

        // Order 777 is processed
        assertTrue(router.processedOrders(777));
    }
}
```

Recommendation

Update the `processedOrders` state in the first loop so subsequent iterations detect the duplicate.

```
// Loop to validate logic and sum total
for (uint256 i = 0; i < len; i++) {
+   if (processedOrders[orderIds[i]]) revert OrderAlreadyProcessed();
+   processedOrders[orderIds[i]] = true;
    _validateOrderLogic(orderIds[i], amounts[i]);
    totalAmount += amounts[i];
}
```

Team Response

Fixed.

[I-01] Batch Safety Cap Check Is Redundant and Can Overflow, Bricking Batch Payments

Severity

Informational Risk

Description

`PaymentRouter.payOrdersBatch(...)` allows users to pay up to 50 orders in one call and enforces a per-order limit via `maxPaymentAmount`. However, after summing the amounts, it performs an additional batch-level check using `maxPaymentAmount * 50`. Because Solidity 0.8+ reverts on arithmetic overflow, if `maxPaymentAmount` is configured above `type(uint256).max / 50`, then `maxPaymentAmount * 50` will revert, causing all batch payments to fail until the limit is reduced. This is primarily an operational risk because `maxPaymentAmount` is set by `MANAGER_ROLE`.

Location of Affected Code

File: `src/PaymentRouter.sol`

```
function payOrdersBatch(
    uint256[] calldata orderIds,
    uint256[] calldata amounts,
    uint256 validUntil,
    bytes calldata signature
) external nonReentrant whenNotPaused {
    // code
    for (uint256 i = 0; i < len; i++) {
        _validateOrderLogic(orderIds[i], amounts[i]);
        totalAmount += amounts[i];
    }

    // @audit Redundant multiplication that can overflow and revert
    if (totalAmount > maxPaymentAmount * 50) revert AmountTooLarge();
    // code
}
```

Impact

A misconfiguration can make `payOrdersBatch(...)` revert for all users until `maxPaymentAmount` is set back to a safe value, causing operational DoS for batch payments. Additionally, the batch check does not add protection beyond `len <= 50` and the per-item check `amounts[i] <= maxPaymentAmount`, meaning it provides no additional safety benefit.

Recommendation

Remove the redundant check, since each order is already validated to be `<= maxPaymentAmount` and the batch length is capped at 50, the batch-level check is redundant and can be safely removed:

```
--- a/src/PaymentRouter.sol
+++ b/src/PaymentRouter.sol

@@ -125,8 +125,6 @@ contract PaymentRouter is ReentrancyGuard, Pausable,
    AccessControl, EIP712 {
      for (uint256 i = 0; i < len; i++) {
          _validateOrderLogic(orderIds[i], amounts[i]);
          totalAmount += amounts[i];
      }
-
-     if (totalAmount > maxPaymentAmount * 50) revert AmountTooLarge();

      // Verify Signature (Batch Hash)
      bytes32 batchHash = keccak256(abi.encode(orderIds, amounts));
```

Team Response

Fixed.

shieldify

Thank you!