Smart Contract Audit

WPay NodeStaking

January 2025

SecHub

CREATED BY

JPG / SUB7 SECURITY

Contents

1 Confidentiality statement

2 Disclaimer

A smart contract security audit is considered a snapshot in time. The findings and recommendations reflect the information gathered during the assessment and not any changes or modifications made outside of that period.

Time-limited engagements do not allow for a full evaluation of all security controls. Sub7 Security prioritized the assessment to identify the weakest security controls an attacker would exploit. Sub7 Security recommends conducting similar assessments on an annual basis by internal or third-party assessors to ensure the continued success of the controls

3 About Sub7

Sub7 is a Web3 Security Agency, offering Smart Contract Auditing Services for blockchain-based projects in the DeFi, Web3 and Metaverse space.

Learn more about us at https://sub7.xyz

4 Project Overview

Wirex Pay represents the future of digital payments, seamlessly integrating blockchain technology with traditional finance. Designed as a modular payment chain and incubated by Wirex, a global leader in the crypto debit card market, Wirex Pay leverages the power of Polygon ZK to offer unparalleled transaction efficiency and security. Supported by Visa, Wirex Pay aligns closely with Visa's vision for the future of payments, emphasizing collaboration in driving fintech innovation.

5 Executive Summary

Sub7 Security has been engaged to what is formally referred to as a Security Audit of Solidity Smart Contracts, a combination of automated and manual assessments in search for vulnerabilities, bugs, unintended outputs, among others inside deployed Smart Contracts.

The goal of such a Security Audit is to assess project code (with any associated specification, and documentation) and provide our clients with a report of potential security-related issues that should be addressed to improve security posture, decrease attack surface and mitigate risk.

As well general recommendations around the methodology and usability of the related project are also included during this activity

1 (One) Security Auditors/Consultants were engaged in this activity.

5.1 Scope

https://github.com/wirexpay/wirex-pay-node-staking.git

5.2 Timeline

From 03 January 2025 to 16 January 2025

5.3 Summary of Findings Identified

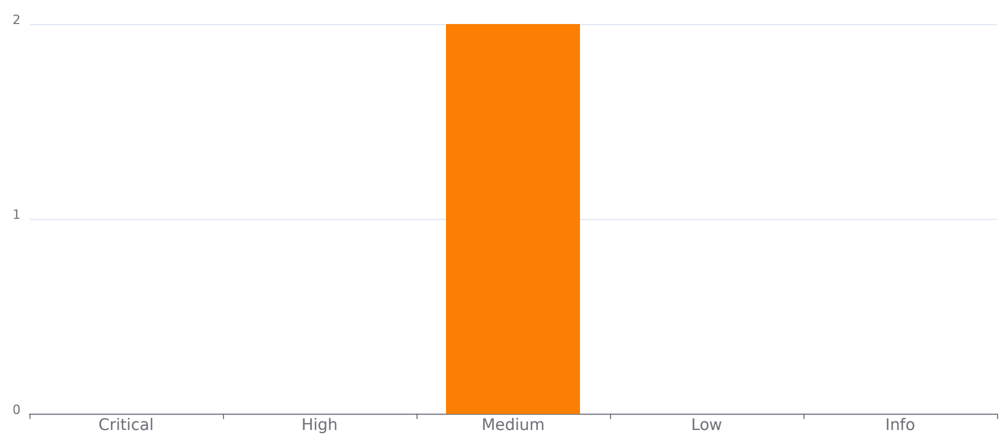

Figure 1: Executive Summary

# 1 Medium Permit Front-Running in depositWithPermit() – ***Fixed***

# 2 Medium Inefficient Reward Distribution Pattern – ***Fixed***

5.4 Methodology

SUB7's audit methodology involves a combination of different assessments that are performed to the provided code, including but not limited to the following:

Specification Check

Manual assessment of the assets, where they are held, who are the actors, privileges of actors, who is allowed to access what and when, trust relationships, threat model, potential attack vectors, scenarios, and mitigations. Well-specified code with standards such as NatSpec is expected to save time.

Documentation Review

Manual review of all and any documentation available, allowing our auditors to save time in inferring the architecture of the project, contract interactions, program constraints, asset flow, actors, threat model, and risk mitigation measures

Automated Assessments

The provided code is submitted via a series of carefully selected tools to automatically determine if the code produces the expected outputs, attempt to highlight possible vulnerabilities within non-running code (Static Analysis), and providing invalid, unexpected, and/or random data as inputs to a running code, looking for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Examples of such tools are Slither, MythX, 4naly3er, Sstan, Natspec-smells, and custom bots built by partners that are actively competing in Code4rena bot races.

Manual Assessments

Manual review of the code in a line-by-line fashion is the only way today to infer and evaluate business logic and application-level constraints which is where a majority of the serious vulnerabilities are being found. This intensive assessment will check business logics, intended functionality, access control & authorization issues, oracle issues, manipulation attempts and multiple others.

Security Consultants make use of checklists such as SCSVS, Solcurity, and their custom notes to ensure every attack vector possible is covered as part of the assessment

6 Findings and Risk Analysis

6.1 Permit Front-Running in depositWithPermit()

Severity: Medium
Status: Fixed

Description

The `depositWithPermit()` function is vulnerable to front-running attacks because it unconditionally executes `permit()` before deposit. An attacker can extract the signature parameters and front-run the transaction by calling `permit()` directly, causing the user's `depositWithPermit()` to revert. As highlighted in EIP-2612 incidents, while front-running permit alone is harmless, when permit is part of a larger transaction sequence (like deposit), it can cause denial of service.

```
// Vulnerable Implementation
function depositWithPermit(...) {
    IERC20Permit(asset()).permit(...); // Can be front-run
    return _depositWithCompound(...);   // Will revert after front-run
}
```

Location

89fc74364dbf3471045e3ce1821e59e848ea2b6b/contracts/TokenizedVault.sol#L139-L149

Recommendation

Implement try-catch to handle front-running gracefully:

```
function depositWithPermit(
    uint256 assets,
    address receiver,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public virtual nonReentrant returns (uint256) {
    try IERC20Permit(asset()).permit(_msgSender(), address(this), assets, deadline, v, r,
        s) {}
    catch {}

    // Check if we have allowance (either from permit or previous approval)
    require(IERC20(asset()).allowance(_msgSender(), address(this)) >= assets, "
        Insufficient allowance");
    return _depositWithCompound(assets, receiver);
}
```

This ensures the deposit can proceed even if permit has been front-run, as long as there's sufficient allowance.

Comments

6.2 Inefficient Reward Distribution Pattern

Severity: Medium
Status: Fixed

Description

The `claimAllRewards()` function implements a non-standard rewards distribution pattern. When there are insufficient rewards tokens, it simply skips distribution instead of sending available balance:

```
// Current Implementation
if (IERC20(vault.asset()).balanceOf(address(this)) < rewardsAmount) {
    return; // Skips distribution entirely when balance < calculated amount
}
```

This deviates from common reward distribution patterns and could result in delayed or lost rewards even when tokens are available to distribute.

Location

c74364dbf3471045e3ce1821e59e848ea2b6b/contracts/WPAYRewardController.sol#L48-L67

Recommendation

When balance is insufficient for full rewards, distribute available balance:

```
function claimAllRewards() public {
    if (!rewardsEnabled || rewardsPeriodStart == 0 || address(vault) == address(0)) {
        return;
    }
    uint256 duration = block.timestamp - rewardsPeriodStart;
    uint256 rewardsAmount = duration * rewardsRate / rewardsPeriod;

    if (IERC20(vault.asset()).balanceOf(address(this)) < rewardsAmount) {
        // Send whatever balance is available
        uint256 balance = IERC20(vault.asset()).balanceOf(address(this));
        rewardsClaimedTotal += balance;
        rewardsPeriodStart = block.timestamp;
        SafeERC20.safeTransfer(IERC20(vault.asset()), address(vault), balance);
        emit RewardClaimed(duration, balance);
        return;
    }

    rewardsClaimedTotal += rewardsAmount;
    rewardsPeriodStart = block.timestamp;
    SafeERC20.safeTransfer(IERC20(vault.asset()), address(vault), rewardsAmount);
    emit RewardClaimed(duration, rewardsAmount);
}
```

This ensures maximum distribution efficiency by sending available rewards rather than skipping distribution entirely.

Comments

SecHub

FOLLOW US