# Security Audit
# Audius Staking Bridge

| | |
|---|---|
| Lead Auditor: | Thomas Lambertz |
| Second Auditor: | Nico Gründel |
| Administrative Lead: | Thomas Lambertz |

November 27$^{th}$ 2023

# Table of Contents

# Executive Summary

**Neodyme** audited two of **Audius'** on-chain programs, namely the **Staking Bridge** and **Payment Router** during late August and September 2023. This report is only for the **Staking Bridge**, with an accompanying report detailing the router.

The auditors found that Audius' programs comprised a clean design and straightforward functionality. According to Neodymes Rating Classification, **one high severity** and only **two informational issues** were found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

All findings were reported to the Audius developers and **addressed promptly**. The security fixes were verified for completeness by Neodyme. In addition to these findings, Neodyme delivered the Audius team a list of nit-picks and additional notes that are not part of this report.

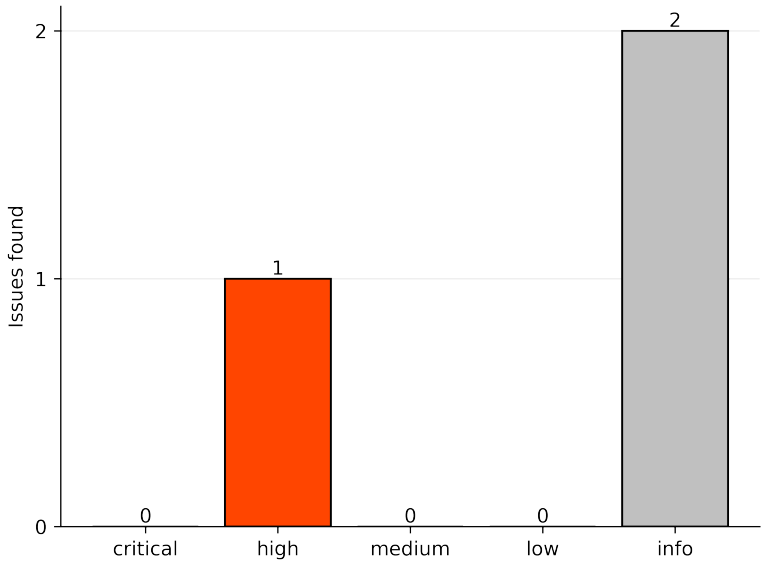


**Figure 1:** Overview of Findings

# 1 Introduction

During the summer of 2023, Audius commissioned Neodyme to conduct a detailed security analysis of two of their new contracts: The Staking Bridge and Payment Router. Two senior auditors performed the audit between August 28th and September 1st, 2023. To account for fixes in the contract, additional auditing was carried out later in September. This report details all findings and their fixes for the **Staking Bridge**.

The audit mainly focused on the contract's technical security but also considered its design and potential social engineering attack vectors. After the introduction, this report details the audit's Scope, gives a brief Overview of the Contract's Design, then goes on to document Findings.

The contracts are straightforward, with no unnecessary authorities or complications. They are open-source and rely on the established Anchor framework, integration tests are available. The Audius team always responded quickly and **competent** to findings and questions of any kind.

## Findings Summary

During the audit, **one security-relevant** and **two informational** findings were identified. Audius remediated all of those findings before the contract's launch.

In total, the audit revealed:

**0** critical • **1** high-severity • **0** medium-severity • **0** low-severity • **2** informational

issues.

The high severity finding addresses a vulnerability where anyone could claim USDC tokens, which the Staking Bridge temporarily held, instead of the intended recipient on Ethereum. All findings are detailed in the Findings section.

# 2 | Scope

The contract audit's scope comprised of a smart-contracts developed by Audius: The **Staking Bridge**. Another contract, the **Payment Router** was audited as well but is not part of this report.

During the audit, we focused on the **implementation** security of the source code and the security of the **overall design**.

All of the source code is located at https://github.com/AudiusProject/audius-protocol/tree/main/solana-programs. Only the bridge contract mentioned is in scope; third-party dependencies are not. As Audius only relies on the well-established Anchor library, the security-txt standard, and a dependency on the hex crate, this does not seem problematic.

Relevant source code revisions are:

- `1659604268cf5f8ac562f05d829d982f2c41c3d1` • Start of the audit
- `e10316718141bd96e683dac57422b5716c70b850` • Commit including all reviewed security fixes

# 3 | Project Overview

This section briefly outlines the contracts' functionality, design, and architecture followed by a discussion on its authorities and security features.

## Functionality

The **Staking Bridge** allows users to easily swap *USDC* to *AUDIO*, which are then transferred via Wormhole to a hardcoded Ethereum address. The swapping and transferring aren't done by users but with a permissionless cranker. This groups multiple user transfers together to reduce fees. There is no on-chain staking-token emitted by the bridge, it is one-way only.

## On-Chain Data and Accounts

The on-chain contract doesn't need to maintain much state. It controls two associated token accounts (ATAs) it can create itself, one for *USDC* and one for *AUDIO* tokens. As authority, it uses a single PDA with seeds `staking_bridge`.

## Authority Structure and Off-Chain Components

There are no explicit contract authorities except for the upgrade authority. Audius plans reqlinquish the upgrade authority before the contracts have substantial adoption. As the contracts don't store any data, they can be redeployed to different addresses if changes are required. This will make the contracts truly permissionless.

There is no configuration or an admin account, as all functions are permissionless. The functionality users gain by staking is handled off-chain and out-of-scope in this audit.

## Instructions

The contract is straightforward and only has four instructions, which we briefly summarize here for completeness.

**Table 1:** Instructions with Descriptions for Staking Bridge

| Instruction | Category | Summary |
|---|---|---|
| CreateStakingBridgeBalancePda | Permissionless, One-Time | Initialize an anchor account at the PDA, which will own the funds. Not strictly necessary to create, but makes it clear which account is the correct one. |
| CreateStakingBridgeBalanceAtas | Permissionless, One-Time | Create two ATAs for the PDA, one for *AUDIO*, one for *USDC*. ATAs can be created by anyone, but this makes it clear the contract only cares about those two tokens. |
| RaydiumSwap | Permissionless | Swap a caller-specified amount of *USDC* tokens from the contracts ATA into *AUDIO* tokens via Raydium. Both token source and recipient account are owned by the hardcoded PDA. The caller specifies slippage parameters. |
| PostWormholeMessage | Permissionless | Transfer a caller-specified amount of *AUDIO* tokens from the contracts ATA to a hardcoded Ethereum address. The caller also specifies which Wormhole nonce to use. |

# 4 | Findings

This section outlines all of our findings. They are classified into one of five severity levels, detailed in Appendix C. In addition to these findings, Neodyme delivered the Audius team a list of nit-picks and additional notes that are not part of this report.

All findings are listed in Table 3 and further described in the following sections.

**Table 2:** Findings

| Identifier | Name | Severity | State |
|---|---|---|---|
| ND-AUD2-H1 | Loss-of-Funds in Staking Bridge Temporary USDC Holdings | High | Resolved |
| ND-AUD2-I1 | Arbitarge and MEV susceptibility | Info | Resolved |
| ND-AUD2-I2 | Anchor readability improvements | Info | Resolved |

## ND-AUD2-H1 – Loss-of-Funds in Staking Bridge Temporary USDC Holdings

| Severity | Impact | Affected Component | Status |
|---|---|---|---|
| **High** | Temporary USDC Holding can be withdrawn by anyone | Swap | Resolved |

A user might use the Payment Router to route USDC payments to the Staking Bridge. The staking-bridge will temporarily hold the funds until a cranker does the Raydium swap to AUDIO and, later, the Wormhole transfer.

The swap between USDC and AUDIO is permissionless; anyone can trigger it. Because of insufficient checks of the used Raydium Pool, an attacker can steal all USDC tokens stored in the contract at that time, by supplying a custom Raydium pool in the swap operation. Raydium allows anyone to create permissionless pools (Docs). Each Raydium Pool has a unique OpenBook DEX Market, but those aren't unique for each token pair.

An attack might look as follows: - Create new OpenBook Market for Audius/USDC - Create a new Raydium Pool for that market - The market has no external liquidity, so that an attacker can manipulate the price at will - Attacker uses `bridge::raydium_swap()` with his own Raydium pool, specifying 1 as `minimum_amount_out` - The contract will sell *all* of the accumulated USDC for 1 audius token. - The attacker withdraws all liquidity from the market, pocketing the gained USDC.

As the contract will hold USDC funds only for short amounts of time, the impact will be limited in terms of total value lost. As the source account is checked to have a USDC mint, the AUDIO funds are not affected.

### Suggestion

Hardcode the Raydium market key or make the swap permissioned. Specifically, we recommend to add the following checks directly in the anchor account macro:

- `program_id` matches Raydium (rename to `raydium_program`)
- `serum_program` matches openbook
- Check that `amm` and optionally `serum_market` match the expected market. Since one is derived from the other, it is technically enough to check only one.

**Remediation**

The Audius team implemented a fix in `65794ebb3fb45442e82f11a48c9bdcd81b40ffee`. The fix implements a check that both the Serum and Raydium markets are correct, which fixes this issue.

## ND-AUD2-I1 – Arbitarge and MEV susceptibility

| Severity | Impact | Affected Component | Status |
|---|---|---|---|
| **Informational** | - | Economics | Remediated |

All on-chain swaps are vulnerable to sandwiching attacks. These are attacks where you sandwich a victim transaction that does a swap between two transactions that manipulate the price of the used market or pool to the attacker's advantage. This is usually defended against by providing a maximum allowed slippage or a target price inside the swap transaction. If the price is outside of that expected range, the swap is aborted.

In the case of the staking-bridge, which uses Raydium, the `minimum_amount_out` value is correctly specified. However, since this instruction is permissionless, an attacker could set this amount arbitrarily low.

How feasible such an attack is depends on many economic factors that are constantly changing. Most important are how liquid the market is, how many funds are available to swap in the staking bridge, and how expensive trading fees and flash-loans are.

If the value of the tokens in the swap is very low compared to the liquidity in the pool, it might not be economical to pay the trading fees to manipulate the price sufficiently. However, the liquidity provider could pull liquidity at any time for free.

### Suggestions

This is an issue with all permissionless swaps and can be worked around in three ways:

1. Make the swap permissioned. The keys authorized to do the swap are then trusted to provide an accurate slippage value.
2. Use a reliable and trusted price oracle. As far as we are aware, that isn't available for AUDIO.
3. Keep the value available for swaps low enough so that it won't be worth it to attack.

It's also possible to do a variant of 1: Make the contract permissioned, but open it up for everyone in case the swap doesn't happen for too long.

### Remediation

The Audius team will ensure that swaps are happening often enough, so that that a sandwich attach won't be economical.

## ND-AUD2-I2 – Anchor readability improvements

| Severity | Impact | Affected Component | Status |
|---|---|---|---|
| **Informational** | - | Anchor Checks | Resolved |

The contract uses Anchor to do a lot of the necessary account checks. All required checks are present, but some could be improved, as they currently are slightly inconsistent. While it does not cause direct security or usability issues, it does make it harder for users to review the contract quickly.

Some improvement suggestions:

- The staking contract currently uses unchecked accounts for the Serum and Raydium programs in `swap()`, and checks them later with a separate function. That is fine, but it's more readable to do that directly in Anchor. Since Raydium isn't an Anchor program, you can't directly rely on Anchors `Program` accounts, but you can put an account key check into the `#[account()]` macro.
- The staking contract currently has inconsistent checks on the accounts that are transparently passed through to Raydium and Serum. Some are verified (like `amm_open_orders`), and some are not (like `pool_coin_token_account`). From a pure security perspective, there don't have to be checks on any of these PDA-derived accounts, as Raydium and Serum will have to do that themselves anyway. This "pre-check" pattern can be helpful for better error messages, but given that the market here should always be the same, it might be unnecessary compute consumption.
- Another example is the `staking_bridge_pda` account. It is initialized as `Account<'info, Empty>`, but later used as `UncheckedAccount<'info>`. While this doesn't cause user-facing issues, it would be more consistent to use the `Empty` tag everywhere. That way, it is also verified that the bump seed provided by the user is the correct one, which isn't guaranteed otherwise. Anchor does *not* verify that the bump seed on an account is the canonical account seed if it is explicitly provided and the account is *not* being initialized. Usually, the account is either being initialized or type-tag-checked. However, since `UncheckedAccount` is used, the type tag check is missing. Due to the way the programs are structured, this doesn't cause visible issues, though. An attacker could swap between a different token-account pair but doesn't gain anything.
- `from_owner` in wormhole doesn't need to be mutable.
- The "CHECK" annotations aren't as helpful as they could be. They can be improved by specifying not only what the account is, but *why* a check is not necessary. Instead of

```
/// CHECK: This is the open orders account for the pool. No check necessary.
```

one could write

```
/// CHECK: This is the open orders account for the pool. Passed through to raydium and checked there.
```

Again, all of these issues aren't security issues but reduce the readability of the code.

**Remediation**

The Audius team improved the checks in a number of commits: `cc4cfab`, `e741112`, `fdf5581`, `308ba2e`, `82f03ac`, `55b8323`.

# A | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe that Neodyme hosts the most qualified auditors for Solana programs. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions, and keep ranking highly among some of the hardest CTF events world-wide. In 2020, some of our members started experimenting with validators and became active members in the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.

# B | Methodology

Neodyme prides itself on not being a checklist auditor. We adapt our approach to each audit, investing considerable time into understanding the program upfront and exploring its expected behavior, edge cases, invariants, and ways in which the latter could be violated. We use our uniquely deep knowledge of Solana internals, and our years-long experience in auditing Solana programs to even find bugs that others miss. We often extend our audit to cover off-chain components, in order to see how users could be tricked or the contract affected by bugs in those components.

Nonetheless, we also have a list of common vulnerability classes, which we always exhaustively look for. We provide a sample of this list below.

- Rule out common classes of Solana contract vulnerabilities, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Solana account confusions
  - Redeployment with cross-instance confusion
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Missing rent exemption assertion
  - Casting truncation
  - Arithmetic over- or underflows
  - Numerical precision errors
- Check for unsafe design which might lead to common vulnerabilities being introduced in the future
- Check for any other, as-of-yet unknown classes of vulnerabilities arising from the structure of the Solana blockchain
- Ensure that the contract logic correctly implements the project specifications
- Examine the code in detail for contract-specific low-level vulnerabilities
- Rule out denial of service attacks
- Rule out economic attacks
- Check for instructions that allow front-running or sandwiching attacks
- Check for rug pull mechanisms or hidden backdoors

# C | Vulnerability Severity Rating

**Critical** Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.

**High** Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.

**Medium** Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable.

**Low** Bugs that do not have a significant immediate impact and could be fixed easily after detection.

**Info** Bugs or inconsistencies that have little to no security impact.

**Neodyme AG**

Dirnismaning 55
Halle 13
85748 Garching
E-Mail: contact@neodyme.io

https://neodyme.io