Security Audit – Sanctum Infinity

Lead Auditor:	Jasper Slusallek
Second Auditor:	Robert Reith
Administrative Lead:	Thomas Lambertz

February 14th 2024

Contents

Executive Summary **4**

1 Introduction **5**
Engagement Details . . . 5
Findings Summary . . . 5
General Assessment . . . 6
Structure of This Report . . . 6

2 Project Overview **7**
Functionality . . . 7
Details on the S Controller . . . 8
Details on the Pricing Program . . . 9
Details on the SOL Value Calculator Programs . . . 9
An Example Swap Flow . . . 10
On-Chain Data and Accounts . . . 11
S controller . . . 11
Flat Fee Pricing Program . . . 13
SOL Value Calculator Programs . . . 13
Instructions . . . 15
S controller . . . 15
Flat Fee Pricing Program . . . 17
SOL Value Calculator Programs . . . 19

3 Scope **20**

4 Findings **21**
[ND-SCT1-H1] State corruption due to faulty memmove destination . . . 22
[ND-SCT1-L1] Adding of LSTs in Flat Fee Pricing Program DOSable . . . 24
[ND-SCT1-I1] LP withdrawal fee can be set to over 100% . . . 26
[ND-SCT1-I2] DOS of initialization of all programs . . . 27
[ND-SCT1-I3] Could require new admin and managers to sign . . . 28

5 Further Discussion **29**
Indirect Smart Contract Risk and Pool Balance . . . 29
Authority Structure . . . 30
Internal Authorities . . . 30

External Authorities 31
Synchronization Issues 32
What are Synchronization Issues? 32
Choosing Fees to Avoid Economic Attacks 33
Rebalance Authority is Exempt from Fee-Based Economic Attack Mitigations 34
Lido's Cranker 34

Appendices

A About Neodyme **35**

B Methodology **36**

C Vulnerability Severity Rating **37**

Executive Summary

Neodyme audited **Sanctum's** on-chain Infinity protocol programs during January of 2024.

The **scope** of the audit was anything relevant to the security of the on-chain contract. In particular, this included **implementation security** of the on-chain contract, resilience to **economic attacks**, correctness of the interaction with external liquid staking contracts, resilience to epoch boundary attacks arising from the interaction with external liquid staking contracts, soundness of the **authority structure**, resilience to attacks from external authorities, as well as any related attack vectors.

The auditors found that Sanctum Infinity, which relies on a new internally developed program framework, was of **very good code quality**. During the audit, it became very clear that the developers were **extremely diligent** in designing the program and making it secure.

According to Neodymes Rating Classification, **no critical vulnerabilities** were found. However, there was **one high-severity issue**, along with a low-severity and some informational findings. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

Note that Sanctum contracted other external audits which concluded before the Neodyme audit. This report excludes any findings that were reported by the other auditors and fixed during the timeframe of the Neodyme audit.

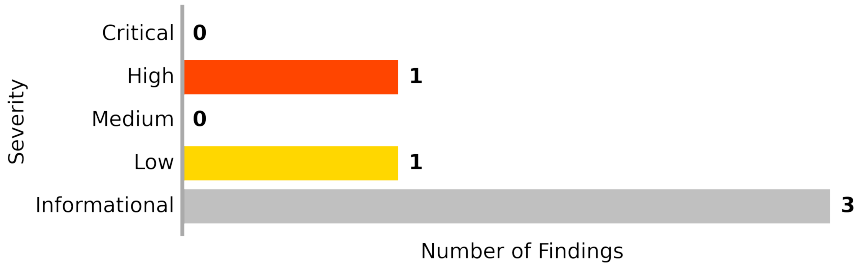

Figure 1: Overview of Findings

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

1 Introduction

Engagement Details

Sanctum engaged Neodyme to do a detailed security analysis of their on-chain Infinity protocol on Solana. This includes the S Controller, which facilitates swaps between different liquid staking tokens (LSTs), as well as the associated pricing and SOL value calculation programs.

Two senior security researchers from Neodyme, Jasper Slusallek and Robert Reith, conducted independent full audits of the contracts between the 6th and the 29th of January 2024. This report contains details on all findings of informational or higher severity that were identified, excluding those that were already being fixed due to reports from prior audits.

Both auditors have a long track record of finding critical and other vulnerabilities in Solana programs. They also have significant experience in auditing both swap protocols and liquid staking protocols. Combined, they've audited or peer reviewed all staking protocols that the Infinity protocol is currently implemented to interact with. Particular care was taken in reviewing these interactions.

Findings Summary

During the audit, **two security-relevant** and **three informational** findings were identified. Sanctum remediated all relevant findings promptly.

In total, the audit revealed:

0 critical • **1** high-severity • **0** medium-severity • **1** low-severity • **3** informational

issues.

The one high-severity finding was on a state corruption vulnerability which would be triggered if the administrator removed an LST from the list of supported LSTs that was not at the start or end of the list. Had the admin done this at any point in time, this would have enabled unprivileged users to exploit funds from the contract. The vulnerability was promptly fixed after it was identified.

All findings are detailed in the Findings Section.

General Assessment

As is evidenced by the fact that the Sanctum developers developed a custom framework, Ideally, for writing secure on-chain contracts, they have **significant experience** in developing and managing Solana programs. This is borne out in the **excellent security, readability and code quality** of the contracts we audited. The programs had **clear structure** and it was obvious that significant thought had been put into all components.

The Sanctum team was both **professional and responsive** in all communications during the audit. The few findings we did have – none of them of critical concern – were **promptly acknowledged and fixed**. For many of the theoretical attack vectors or improvements we thought about during the audit, it turned out that they had already discussed them internally or outlined them in their documentation. We were impressed by how much thought was put into these considerations.

Structure of This Report

After this introduction, this report will give a Project Overview detailing the functionality, on-chain state and instructions of the various contracts, as well as give an illustrative Example of a protocol interaction.

Afterward, we give details on the audit's Scope and elucidate on the Findings that were identified.

Finally, we discuss further topics such as the Indirect Smart Contract Risk and the Authority Structure of the contract, as well as attack vectors that arise from Synchronization Issues.

2 | Project Overview

Functionality

Sanctum's Infinity protocol, or Sanctum v2, provides a shared liquidity pool for swapping between different liquid staking tokens (LSTs). Users can swap between the LSTs at a price corresponding to the ratio of their intrinsic SOL valuations, as determined from the on-chain state of the corresponding external liquid staking program.

Swaps from one LST to another are possible as long as the pool's reserve of the destination LST has funds. To maintain this liquidity, users can deposit any LST in exchange for LP tokens which represent a share of the total SOL value of all LSTs that the pool holds. They can later burn these LP tokens for a correspondingly valued share of any other LST of their choice. There is also a mechanism to balance liquidity between the pool's reserves, which we will discuss further below.

Sanctum v2 is split into three major components: The S Controller, the Pricing Program and the SOL Value Calculator Programs. See Figure 2 for an overview of their interactions, both internally and externally.

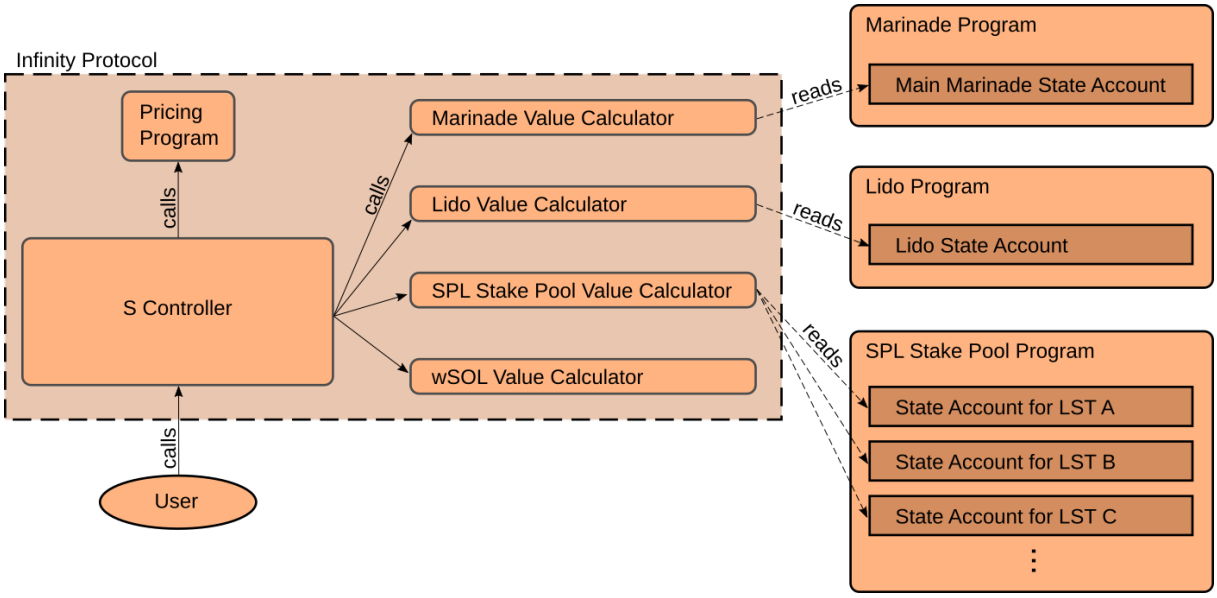

Figure 2: Overview of programs of the Infinity protocol, including their internal and external interactions. The S Controller calls the Pricing Program and SOL Value Calculator Programs to correctly price swaps and liquidity operations. The SOL Value Calculator Programs read the pricing data from the respective external liquid staking program's state accounts. Note that the wSOL SOL Value Calculator Program does not read any external data, as 1 wSOL is valued at exactly 1 SOL.

The **S Controller** is the main component of Sanctum v2. It holds the LST reserves, stores and continually updates the SOL valuation of these reserves, and handles both swapping of LSTs and depositing or removing of liquidity.

In doing these, it employs the help of a **Pricing Program**, which calculates fees for swapping between LSTs or for removing liquidity, and potentially for providing liquidity. There are multiple possible fee structures which can be implemented in different programs. The S Controller can seamlessly switch between which of these programs it uses.

The S Controller also employs the help of the **SOL Value Calculator Programs**. There is one such program for each supported LST. It uses data from the external liquid staking protocol that controls the LST to calculate the intrinsic SOL value that the LST has.

We describe each of these components along with their functionality in more detail in the following sections.

Details on the S Controller

The S Controller only has four user-facing instructions: Two for swapping between LSTs where they specify either the input or the output amount, and two for adding or removing liquidity by depositing or withdrawing a share of one of the pool's LST reserves. However, there is significant functionality in the background for synchronizing the value of the LST reserves, keeping them balanced, and administrating the pool.

The most important of these is the synchronization of the value of the LST reserves. The pool must, at all times, keep an accurate record of the total value of its LST holdings in order to calculate the value of LP tokens when users deposit or remove liquidity. To do this, the pool exposes a permissionless instruction which acts on one of the LSTs and synchronizes the internally saved value of the that LST's reserves with the intrinsic value of the LST, as determined by the corresponding SOL value calculator program.

For keeping the reserves of the different LSTs balanced, the S Controller has a rebalancing mechanism where a rebalance authority can flash loan a share of the reserves of one type of LST to externally swap it to an equally- (or greater-)valued share in other LSTs. In doing this, the Controller ensures that the total SOL value that the pool holds does not decrease when rebalancing.

The S Controller levies fees for swapping between LSTs, as well as for adding and removing liquidity with the help of the Pricing Programs. The Flat Fee Pricing Program, which seems to be the one intended for mainnet use, however, charges zero fees for adding liquidity. Fees are split between the reserves – thus contributing to the appreciation of the LP token – and the protocol fee vaults, which are controlled by the S Controller's fee beneficiary authority.

On the administrative side, aside from basic functionality, there is a mechanism for phasing out an LST by forbidding all actions that would increase that LST's reserves. Finally, there is also a security mechanism to pause all pool operations, e.g. in case of an incident with one of the supported LSTs. The latter can be triggered by any authority from a list that the contract stores internally. This pausing can only be reversed by the administrator of the protocol.

Details on the Pricing Program

The Pricing Program calculates the fees that the S Controller charges for swaps and for adding or removing liquidity.

For each of these actions, it exposes a permissionless function that anyone can call in order to calculate the fees associated with that action. It does not touch any funds, instead getting an input amount as an integer and returning an output amount as an integer using Solana's little-used CPI return value functionality. Note that the pricing program deals exclusively with values denominated in SOL, so that all LST pricing and conversion logic is instead done in the SOL Value Calculator Programs.

There are currently two Pricing Programs implemented, the Flat Fee Pricing Program and the No Fee Pricing Program.

The Flat Fee Pricing Program, which seems to be planned for mainnet use, has a two fees associated with each LST – the input and output fee – as well as a global fee for removing liquidity; all fees are linear in the amount involved and measured in basis points. When swapping from LST A to LST B, the S Controller calls the program with the SOL value of the LST position that the user wishes to swap. The program then adds the input fee basis points of LST A and output fee basis points of LST B together and applies the resulting fee to the input SOL value, returning the remainder after application of the fee to the S Controller. For removing liquidity, the input SOL value is simply reduced by the flat fee and the result is returned.

The No Fee Pricing Program, true to its name, does not charge any fees. Instead, for all actions, it simply returns any input values that are given to it.

Details on the SOL Value Calculator Programs

There is one SOL value calculator program for each LST. It's called by the S Controller to convert an LST amount to its intrinsic SOL value, or vice-versa. The intrinsic SOL value is the amount of SOL that users would receive if they traded in their LST for SOL using the functionality of the external program which manages the respective LST, respecting any fees that the program charges. This value is calculated by inspecting the on-chain state accounts of the external program and mirroring the program's internal pricing calculation.

Currently, there exists a SOL value calculator program for Marinade (mSOL), Lido (stSOL) and the SPL stake pool program (any tokens created using it, including jitoSOL, bSOL, laineSOL etc.). There also exists a program for wrapped SOL (wSOL) which simply acts as an identity function for any conversions.

An Example Swap Flow

To better understand both the architecture of the Infinity protocol and well as where potential security flaws may lie, we go through what exactly happens if a user uses Infinity to swap from one LST to another, specifying the input amount.

Abstracting away a few details, the general flow is as follows:

1. User calls the S Controller, saying they want to swap `amount_in` LST A to LST B
2. The S Controller does a CPI into the SOL Value Calculator Program for LST A to determine what the intrinsic SOL value of `amount_in` LST A is. The calculator program responds by returning `amount_in_sol`.
3. The S Controller does a CPI into the Flat Fee Pricing Program to determine how many fees are charged on a swap from LST A to LST B with SOL value `amount_in_sol`
    i. The Flat Fee Pricing Program fetches the input fee `fee_a_in_bps` of LST A and output fee `fee_b_out_bps` of LST B from its own on-chain state
    ii. The two fees are added and applied to `amount_in_sol`. The remainder, `amount_out_sol`, is returned.
4. The S Controller does a CPI into the SOL Value Calculator Program for LST B to determine what amount of LST B has an intrinsic SOL value of `amount_out_sol`. The calculator program responds by returning `amount_out`.
5. `amount_in` LST A is transferred from the user to the protocol's LST A reserves.
6. The total swap fees, as calculated by the Pricing Program, are `amount_in_sol - amount_out_sol`. A fixed percentage of these fees is declared as protocol fees, converted from SOL to LST B using the valuation from step 4, and transferred from the LST B reserves into the LST B fee vault. The rest of the fees stays in the reserves, hence rewarding liquidity providers.
7. `amount_out` LST B is transferred from the LST B reserves to the user.

Some checks are omitted, e.g. that the user receives at least the amount of LST B that they want and that the pool's total SOL value did not decrease.

See Figure 3 for a visual representation.

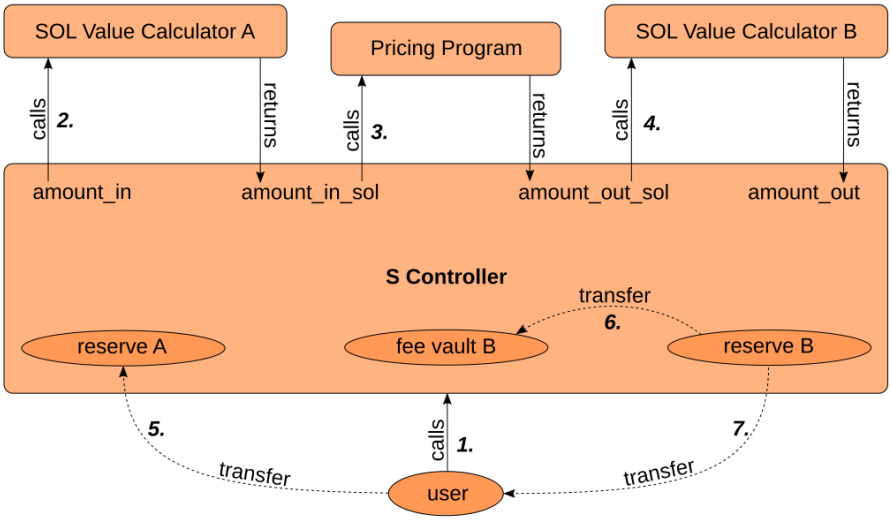

Figure 3: A visualization of the example swap flow.

On-Chain Data and Accounts

We give a brief overview of what data the contracts store on-chain and what accounts/PDAs they use.

S controller

The S controller maintains multiple PDAs to keep track if its state and parameters. The following data needs to be tracked:

- The configuration parameters, in particular
  - the share of fees that go to the protocol,
  - the address of the global pricing program,
  - the mint account of the LP token, and
  - the current protocol version.
- Authorities, in particular
  - the admin,
  - the fee beneficiary,

  - the rebalance authority, and
  - the authorities that can trigger pausing of the pool (the "DisablePool authorities").
- Global state information, in particular
  - the total SOL value of all pool assets,
  - whether the pool is currently paused ("disabled"), and
  - whether the pool is currently in the process of being rebalanced.
- Relevant information on each LST, in particular
  - its mint,
  - the address of its SOL value calculator program,
  - the SOL value of pool reserves of this LST, and
  - whether this LST's reserves are currently in reduce-only mode ("input disabled").
- Information on the currently happening rebalance, if any; in particular, this is
  - the destination LST of the rebalance, and
  - the total SOL value of the pool before the rebalance.

This data is represented on-chain in PDAs. They are structured as follows:

Pool State PDA This is the main state account of the protocol. Only one such PDA exists at the fixed address derived from the seed `[b"state"]`. It stores the configuration parameters, the addresses of all authorities except the pause authorities, and the global state information.

DisablePool Authority List PDA This PDA, derived from the fixed seed `[b"disable-pool-authority-list"]`, stores the pause authorities as a list of public keys. A signature from any single key on this list is enough to trigger pausing of the pool.

LST State List PDA This PDA, derived from the fixed seed `[b"lst-state-list"]`, stores a list with one entry for each LST. It contains the relevant information on the LSTs, as listed above.

RebalanceRecord PDA This PDA at the fixed seed `[b"rebalance-record"]` stores information on a rebalance that is currently happening. The account only exists during the rebalance and is closed as soon as the rebalance concludes. Since a rebalance must conclude in the same transaction that it is started, this PDA never exists outside of a transaction.

Additionally, the following PDAs are used as authorities:

Pool State PDA (again) Besides storing the contract's data, the pool state PDA is also used as an authority on the LP token mint and on the reserves of the pool are also ATAs of the Pool State PDA, hence it must sign for any LP token mints and transfers of funds from the reserves.

Protocol Fee Authority PDA This PDA is derived from the fixed seed `[b"protocol-fee"]`. It is the token account owner of all Protocol Fee Accumulator ATAs, as specified below, and therefore must sign when withdrawing fees.

Finally, the contract maintains the following token accounts and mints to manage the contract's funds:

LST Reserve ATAs The reserve for each LST is the ATA derived from the Pool State PDA (along with the LST's mint and token program).

Protocol Fee Accumulator ATAs Additionally, each LST has an ATA that collects the fees levied in that LST (e.g. when removing that LST as liquidity, or swapping to that LST). It is the ATA derived from the Protocol Fee Authority PDA (along with the LST's mint and token program).

LP Token Mint This is a mint account that's given to the Infinity protocol during initialization. As the name suggests, it is the mint of the token that users receive for providing liquidity. The address of the mint is stored in the Pool State PDA on initialization and verified whenever it is used. The minting and freezing authority is set to the Pool State PDA.

Flat Fee Pricing Program

Note that the No Fee Pricing Program does not maintain on-chain state, as it simply always returns any input values. The Flat Fee Pricing Program, on the other hand, maintains on-chain state using PDAs. The following data must be stored:

- the program's manager,
- the fee for removing liquidity, and
- for each LST, its input and output fee. That is, the portion of fees that are charged when that LST is the source or destination, respectively, of a swap.

This data is stored in the following PDAs:

Program State PDA This PDA, derived from the fixed seed `[b"state"]`, stores the program manager and the LP withdrawal fees.

Fee Account PDA There is one such PDA for each LST, derived from the seed `[b"fee", lst_mint_address]`. It stores the input and output fee associated with that LST.

The Flat Fee Pricing Program does not maintain any authority PDAs and does not manage funds.

SOL Value Calculator Programs

All SOL value calculator programs have the same internal state structure. They must store:

- the program's manager, and
- the slot at which the corresponding external program was last upgraded.

These are both stored in the:

Calculator State PDA A PDA at the fixed address `[b"state"]` that stores the manager and last upgrade slot.

The programs have no authority PDAs and do not handle funds.

However, all calculator programs also read on-chain state from the corresponding external programs. The following sections contain details on this for each calculator program. Note that for the wSOL program, no state is needed, since the calculator program simply returns the input values it is given.

External State: Marinade

For the Marinade SOL value calculator program, the program reads data from the main marinade state account, which resides at `8szGkuLTAux9XMgZ2vtY39jVSowEcpBfFfD8hXSEqdGC`.

From there, it reads all parameters needed for calculating the total SOL under control and available to mSOL holders (calculated as `(total_active_balance + delayed_unstake_cooling_down + emergency_cooling_down + available_reserve_balance - circulating_ticket_balance)`), as well as the `msol_supply`. Furthermore, it reads the delayed unstake fee `delayed_unstake_fee_bp_cents` in order to factor it in when calculating the intrinsic SOL valuation of mSOL.

External State: Lido

For the Lido SOL value calculator program, the program reads data from the Lido state account, which resides at `49Yi1TKkNyYjPAFdR9LBvoHcUjuPX4Df5T5yv39w2XTn`.

The calculator program simply reads `sol_balance`, the total SOL under Lido's control, and the `stsol_supply`. These suffice to calculate the intrinsic SOL valuation of stSOL. No fees need to be considered, since Lido does not charge fees for trading stSOL into SOL (i.e. withdrawing a stake account).

External State: SPL Stake Pool

For the SPL Stake Pool SOL value calculator program, note that the SPL stake pool program administrates multiple LSTs. Each LST it manages has a different state account. The calculator program can read data from any such state account, while verifying that it gets the correct one for the LST it is supposed to be operating on.

The calculator program reads `total_lamports`, the total SOL under control for that LST and available to its holders and the `pool_token_supply`. Furthermore, it reads the two fee parameters `stake_withdrawal_fee_numerator` and `stake_withdrawal_fee_denominator` in order to factor in the fee in the intrinsic SOL value calculation.

Instructions

For completeness, we provide brief descriptions of the instructions that the contracts expose, along with their functionality.

S controller

First, the S controller, which has a total of 21 instructions. Note that most of them are privileged and that only swap operations, liquidity operations and the synchronization instruction are permissionless.

Instruction	Category	Summary
Initialize	Init Authority Only	Must be called by a hardcoded initial authority. Initializes the contract by creating the pool state account and initializing the LP token's mint account along with its token extensions.
AddLst	Admin Only	Adds an LST by adding a new LSTState to the LSTState list. Creates ATAs for the pool reserves and fee accumulator.
RemoveLst	Admin Only	Removes an LST from the LSTState list.
DisablePool	Any DisablePool Auth	Must be called by any authority on the DisablePool list. Disables all user-facing and also most privileged instructions.
EnablePool	Admin Only	Reenables the pool.
AddDisablePoolAuthority	Admin Only	Adds an authority to the list of authorities that can call DisablePool.
RemoveDisablePoolAuthority	Admin or Key Itself	Must be called by admin or the key being removed from the list. Removes the authority from the DisablePool list.
SetAdmin	Admin Only	Sets the admin authority to a new key
SetPricingProgram	Admin Only	Sets a new pricing program
SetSOLValue-CalculatorProgram	Admin Only	Sets a new SOL value calculator program for a given LST
SetProtocolFeeBeneficiary	Fee Beneficiary Only	Sets a new Fee Beneficiary

Instruction	Category	Summary
SetProtocolFee	Admin Only	Sets the share of fees that the protocol takes for itself (or rather, for the fee beneficiary)
SetRebalanceAuthority	Admin or Rebalancer	Must be called by admin, or by the old rebalance authority; Sets a new rebalance authority
WithdrawProtocolFees	Fee Beneficiary Only	Withdraws an arbitrary amount from one of the protocol's fee vaults.
StartRebalance	Rebalancer Only	Withdraws some amount of a given LST from the pool's reserves and gives it to the rebalance authority for external trading. Ensures that a corresponding EndRebalance Instruction exists in the same transaction. During the Rebalance, the Rebalancer will need to deposit other LSTs in order for the EndRebalance valuation check to succeed.
EndRebalance	Rebalancer Only	Checks that the total SOL value of the pool did not decrease since before the StartRebalance instruction.
SyncSolValue	Permissionless	Synchronizes the pool's stored value of a given LST's reserves with a new intrinsic SOL value of the LST, as calculated from the on-chain state of the corresponding external liquid staking program
AddLiquidity	Permissionless	Allows users to add liquidity by depositing an arbitrary supported LST. The users receive an amount of LP tokens proportional to the SOL value they added to the pool in relation to the total SOL value the pool already has.

Instruction	Category	Summary
RemoveLiquidity	Permissionless	Allows users to remove liquidity by burning their LP tokens in exchange for a share of an arbitrary LST. The amount of the LST they get depend on the share of LP tokens they are burning, the total SOL value of all pools, and the current intrinsic SOL valuation of the LST they want to withdraw. Levies fees in the LST being withdrawn.
SwapExactIn	Permissionless	Allows users to swap from any LST to any other LST. They specify the input amount for the swap. The price of the swap is determined by the current intrinsic SOL value of the LSTs via the SOL value calculator programs. Fees are calculated by the global pricing program. The protocol takes a fixed proportion of the fees, the rest remains in the pool and contributes to LP token appreciation.
SwapExactOut	Permissionless	Same as SwapExactIn, but the user specifies the output amount of the swap, not the input.

Flat Fee Pricing Program

We continue with the Flat Fee Program, which is the fee program intended for mainnet use. As noted before, a No Fee Program is also implemented, which only exposes the PriceExactIn, PriceExactOut, PriceLpTokensToMint and PriceLpTokensToRedeem instructions, and all of them simply return the input values they are given.

Instruction	Category	Summary
Initialize	Permissionless	Can only be called once. Initializes the program by creating and populating the state PDA.
AddLst	Manager Only	Adds support for an LST by creating and populating its associated fee account PDA.

Instruction	Category	Summary
RemoveLst	Manager Only	Removes support for an LST by closing its fee account PDA.
SetManager	Manager Only	Sets a new manager.
SetLstFee	Manager Only	Sets a new input and output fee for an LST in its fee account PDA.
SetLpWithdrawalFee	Manager Only	Sets a new LP withdrawal fee in the state PDA.
PriceExactIn	Permissionless	Calculates the SOL value of the fees levied for a swap from one given LST to another given LST by adding the respective input and output fee. Returns the remainder after application of the fee.
PriceExactOut	Permissionless	The same as PriceExactIn, but now it is given the target SOL value of the swap, and calculates the necessary input before application of the fee.
PriceLpTokensToMint	Permissionless	Normally, this would be given the SOL value of a liquidity deposit, calculate the fee levied on it and return the remainder after application of the fee. However, in the flat fee pricing program, no fees are levied on depositing liquidity, hence any values given to this instruction are simply returned unchanged.
PriceLpTokensToRedeem	Permissionless	Given a SOL value of a liquidity removal operation, calculate the fee levied on it and return the remaining SOL value after application of the fee.

SOL Value Calculator Programs

Finally, we come to the SOL value calculator programs. The calculator program of each LST exposes the same instructions as listed below. The only exception to this is wSOL, which though it is treated as an LST for convenience, simply wraps SOL. As such, it only exposes the LstToSol and SolToLst instructions, and they simply return all values given to them unchanged.

Instruction	Category	Summary
Init	Permissionless	Can only be called once. Initializes the program by creating and populating the state PDA.
SetManager	Manager Only	Sets a new manager.
UpdateLastUpgradeSlot	Manager Only	Reads the last program upgrade slot from the program data account of the external liquid staking program, and updates it in the state PDA.
LstToSOL	Permissionless	Given an LST amount, calculates its SOL value. The LST value is calculated via data read from the external program's on-chain state. Cannot be executed if the last upgrade slot stored in the state PDA does not match the external program's. Additional restrictions apply for some programs – e.g. for Lido, its exchange rate must have already been updated this epoch, and for marinade, the pool must not be paused.
SolToLst	Permissionless	Given a SOL amount, calculates the equivalent amount of LST using the same valuation as LstToSol. The same additional restrictions apply.

3 | Scope

The audit's scope comprised major factors relevant to the on-chain program's security. In particular, this included:

- **Implementation** security of the source code
- Security of the **overall design**
- Correctness of the **interaction with external liquid staking contracts**
- Resilience again **economic attacks**, in particular those arising from the effects of epoch boundaries on the external staking contracts
- The contract's **authority structure**, including the powers of external authorities

All of the source code of the on-chain program was in scope of this audit. However, third-party dependencies were not. Note that the source code is currently closed source, and as such we cannot link to the exact revisions or to the code locations of findings.

Toward the start and end of the audit, several fixes from other audits were pushed. We have excluded from our report any vulnerabilities that were fixed in the course of this.

Relevant source code revisions are:

- `dd0768d496fdba80a019ccce86d187fbc2355223` • Start of the audit
- `84acfb648e0f9aa676ae4cb9fd9480fc1a989638` • Last reviewed revision

4 | Findings

This section contains details on all findings of at least informational severity found during the audit. Findings which were fixed during the review due to reports from prior audits were excluded. The vulnerabilities are classified according to Neodyme's Severity Rating. In total, there were:

- **0** critical issues,
- **1** high-severity issues,
- **0** medium-severity issues,
- **1** low-severity issues, and
- **3** informational issues.

Below is the full list of vulnerabilities along with their severity and resolution status.

Name	Severity	Status
[ND-SCT1-H1] State corruption due to faulty memmove destination	High	Resolved
[ND-SCT1-L1] Adding of LSTs in Flat Fee Pricing Program DOSable	Low	Resolved
[ND-SCT1-I1] LP withdrawal fee can be set to over 100%	Info	Resolved
[ND-SCT1-I2] DOS of initialization of all programs	Info	Resolved
[ND-SCT1-I3] Could require new admin and managers to sign	Info	Acknowl.

[ND-SCT1-H1] State corruption due to faulty memmove destination

Severity	Impact	Affected Component	Status
High	Enables loss of funds, likely to be accidentally triggered by admin	RemoveLST and RemoveDisable-PoolAuthority	Resolved

In `remove_from_list_pda`, the function tries to remove a list item by copying the data from all items after it one to the left. However, the implementation (after simplification) actually copies entries from the range `data[index+1 .. array_size]` to `data[0 .. array_size-(index+1)]`, as can be seen in the following code snippet.

```
pub fn remove_from_list_pda<T: AnyBitPattern>(
    RemoveFromListPdaAccounts {
        list_pda,
        refund_rent_to,
    }: RemoveFromListPdaAccounts,
    index: usize,
) -> Result<(), ProgramError> {
    // shift [index+1..] items left to overwrite [index]
    let index_plus_one_byte_offset = index
        .checked_add(1)
        .and_then(|i_plus_1|
        i_plus_1.checked_mul(std::mem::size_of::<T>()))
        .ok_or(SControllerError::MathError)?;
    let remaining_byte_count = list_pda
        .data_len()
        .checked_sub(index_plus_one_byte_offset)
        .ok_or(SControllerError::MathError)?;
    unsafe {
        let mut data = list_pda.try_borrow_mut_data()?;
        let index_ptr = data.as_mut_ptr();
        std::ptr::copy(
            index_ptr.add(index_plus_one_byte_offset),
            index_ptr, // <- incorrect destination!
```

```
            remaining_byte_count,
        );
    }

    // [...]

    Ok(())
}
```

This is incorrect, since it just copies the slice of all items after the item to be deleted to the front of the array, instead of one to the left.

This means that the function overwrites entries that it did not want to delete (instead creating duplicate entries) and potentially does not even delete the entry it did want to delete. This will result in corrupted state for any deletion which isn't the first or last item.

The function is used in two places: When removing an LST and when removing a DisablePool authority. The latter case isn't too bad since it just temporarily duplicates and removes some authorities from the list, which the admin can recover. The former case is critical, since it it means that anyone could, in the meantime, exploit the Infinity protocol by calling SyncSolValue on the duplicate entries and exploiting the resulting incorrect total SOL value of the pool via liquidity operations. It is also difficult to recover, since it requires upgrading to recover the LstStates from before the deletion.

Resolution

The Sanctum engineers patched the function to correctly copy the entries to `data.as_mut_ptr() + index*size(T)` (with checked math). Neodyme verified the fix.

[ND-SCT1-L1] Adding of LSTs in Flat Fee Pricing Program DOSable

Severity	Impact	Affected Component	Status
Low	Inability to add new LSTs	AddLst in Flat Fee Pricing Program	Resolved

The AddLst instruction in the Flat Fee Pricing Program uses `create_account` to create the Fee Account PDA, as can be seen in the following code snippets:

```
pub fn process_add_lst(accounts: &[AccountInfo], args: AddLstIxArgs)
-> ProgramResult {
    // [...]

    create_rent_exempt_account_invoke_signed(
        CreateAccountAccounts {
            from: payer,
            to: fee_acc,
        },
        CreateRentExemptAccountArgs {
            space: program::FEE_ACCOUNT_SIZE,
            owner: program::ID,
        },
        &[create_pda_args.to_signer_seeds().as_slice()],
    )?;

    // [...]

    Ok(())
}
```

```
pub fn create_rent_exempt_account_invoke_signed(
    accounts: CreateAccountAccounts,
    args: CreateRentExemptAccountArgs,
    signer_seeds: &[&[&[u8]]],
) -> ProgramResult {
```

```
    let args = args.try_calc_lamports_onchain()?;
    create_account_invoke_signed(accounts, args, signer_seeds)
}
```

```
pub fn create_account_invoke_signed(
    accounts: CreateAccountAccounts,
    args: CreateAccountArgs,
    signer_seeds: &[&[&[u8]]],
) -> ProgramResult {
    let ix = create_account_ix(CreateAccountKeys::from(accounts),
    args); // <- builds a DOSable create_account instruction
    let account_infos: [AccountInfo; CREATE_ACCOUNT_ACCOUNTS_LEN] =
    accounts.into();
    invoke_signed(&ix, &account_infos, signer_seeds)
}
```

This fails if the account already has a non-zero amount of lamports. Anyone can cause this by simply sending lamports to the PDA.

This means anyone can prevent the adding of a new LST. The only way to resolve the situation is a contract upgrade, or deploying and switching to a different pricing program.

Resolution

Sanctum corrected this by instead creating the account via the system program's `allocate` and `assign` instructions, which is the standard way to mitigate this vulnerability. Neodyme verified the fix.

[ND-SCT1-I1] LP withdrawal fee can be set to over 100%

Severity	Impact	Affected Component	Status
Info	Admin can set fees over 100%	SetLpWithdrawalFee in Flat Fee Pricing Program	Resolved

In the SetLpWithdrawalFee in the Flat Fee Pricing Program, the new fee, which stores the fee in basis points, is an unconstrained `u16`. Only the admin can call this instruction. However, it may be prudent to check that this fee is between 0% and 100%.

Note that if the combined fee of a swap is at over 100%, this will fail the swap. Hence this has little impact on users, except for DOSing swaps.

Resolution

The Sanctum developers added an upper bound check for the percentage to the instruction. Neodyme verified the fix.

[ND-SCT1-I2] DOS of initialization of all programs

Severity	Impact	Affected Component	Status
Info	DOS of initialization	Initialization instruction in all programs	Resolved

The Initialize instructions of all programs also use `create_account` to create their main state account. This can be DOSed by sending lamports to the account beforehand. An attacker could theoretically do this if they know the account where the state account address will reside beforehand.

This finding is not particularly interesting since you can just deploy at another account, but it is not hard to fix either.

Resolution

Again, Sanctum corrected this by instead creating the account via the system program's `allocate` and `assign` instructions, which is the standard way to mitigate this vulnerability. Neodyme verified the fix.

[ND-SCT1-I3] Could require new admin and managers to sign

Severity	Impact	Affected Component	Status
Info	Possible loss of control over authority	Authority changing instructions in all programs	Acknowledged

When setting a new admin in the S controller, or when changing the manager in the pricing of SOL value calculator programs, the new admin/manager does not have to sign. It may be prudent to add this check to prevent accidents such as typos, copy-paste errors or similar.

Resolution

Sanctum acknowledged the finding, but stated that they would keep the old version for ease of setting these authorities to multisig addresses.

5 | Further Discussion

In this section, we discuss further aspects of the Infinity protocol programs' security. In particular, this includes the indirect smart contract risk and balancing issues that the Infinity protocol is exposed to due to its interaction with external liquid staking programs, the structure of the protocol's authorities and what would happen if they were compromised, as well as an analysis of attack vectors related to price synchronization issues.

Indirect Smart Contract Risk and Pool Balance

Any incident where the market price of an LST diverges non-negligibly from its intrinsic value presents a risk for liquidity providers. In an efficient market, all liquidity in the pool will be traded into or out of the affected LST, leaving liquidity providers with only the LST(s) that are worth less. In these situations, the rebalance authority cannot resolve the situation by rebalancing the pool without making a loss in their external trades.

One of the principal cases where this can happen is a security incident in one of the external liquid staking protocols. Indeed, the interaction with these protocols means that the Infinity protocol carries significant indirect smart contract risk. A bug in the internal tracking of funds or the minting of LST tokens of any of these protocols means that potentially, all of the Infinity protocol's liquidity is at risk. We recommend automated monitoring to detect any incidents, though this is only a partial remedy.

However, price divergence need not be directly caused by a pricing bug. Another risk is a temporary or even permanent DOS of the external protocol, which leaves the Infinity protocol with LST tokens that cannot be redeemed until the DOS is resolved, if ever.

We mention that for the three relevant external liquid staking programs that are currently supported – Marinade, Lido and the SPL Stake Pool Program – Neodyme has either audited or peer-reviewed them. We would estimate that the likelihood of bugs that would cause such significant mispricings is low. We would, however, caution that care must be taken with the integration of Lido due to its shutdown and unmaintained status on Solana; it is not guaranteed that Lido would react to or resolve any security incidents of the Solana contract.

Outside of security incidents, other changes in market valuation of LSTs, e.g. due to the external protocols changing their fee share of staking rewards, or the liquidity pool of an external program being empty, invite toxic flow into the Infinity protocol. Care must be taken in choosing fees such that Infinity protocol liquidity providers still have upside and such that the rebalance authority does not have to subsidize frequent rebalancing of the pool.

Finally, there is also the case that one of the external programs' authorities is compromised. We discuss the risk of this, among other things, in the following subsection.

Authority Structure

In this section, we break down the different authorities that are relevant to the Infinity protocol – both internal and external – and discuss what would happen if they were compromised. We also discuss how they are managed or planned to be managed.

Internal Authorities

The authorities that the Infinity protocol uses are split into those of the S controller, the Pricing Program and the SOL value calculator programs.

The S controller employs the most authorities: The upgrade authority, the admin, the rebalance authority, the DisablePool authorities and the fee beneficiary.

The **S controller upgrade authority** has total control over the entire contract. As in any contract, it can arbitrarily add or replace functionality and can therefore also steal all liquidity. It is the highest authority in the contract and should be managed accordingly. Sanctum stated that they plan to employ their multisig for the upgrade authority upon launch.

The **admin authority** can add and remove LSTs, disable the pool or LST inputs, set fees, arbitrarily set all other authorities except the upgrade authority and fee beneficiary, and set the pricing program and SOL value calculator programs. Of these, adding LSTs and setting the SOL value calculator or pricing programs are the most critical. By, for example, temporarily setting the SOL value calculator program of an LST to one that returns incorrect prices, they can also steal all liquidity from the protocol. Sanctum has stated that they intend to have the admin as a hot wallet during setup of the contract, and may move it to a hardware wallet when there is less low-level administrative work to be done. We advised that using a hardware wallet or preferably a multisig for this authority is highly recommended.

The **rebalance authority** can execute rebalances between different LSTs. However, it is subject to the restriction that all rebalances must not decrease the total SOL valuation of the pool. As such, it has very little power. However, as we mention in the section on the rebalance authority in the section on synchronization issues, it can profit from sandwiching pricing updates of the external liquid staking programs. This is, however, only of informational severity. Sanctum has stated that the rebalance authority will remain with them, likely as a hot wallet operated by a bot.

The **DisablePool authorities** have the power to pause (but not unpause) the protocol. This power lies with any single account in the DisablePool authority list. This pausing completely disables all

user interactions with the pool, including withdrawing liquidity. Only the admin can unpause the contract.

The **fee beneficiary** can only withdraw fees from the fee vaults. Sanctum stated that it will be controlled by their multisig, and that it may move into the control of a DAO if one is launched in the future.

The Flat Fee Pricing Program only has two authorities: The upgrade authority, and the manager.

The **pricing upgrade authority** can replace all functionality of the contract. This is also critical, since by returning bogus values, the pricing program can set swap prices arbitrarily. Sanctum has stated that the authority will be managed by the Sanctum multisig.

The **pricing manager** has limited power, in that they can set LST fees within fixed bounds defined by the program. At worst, a compromise of this authority would lead to no fees or very high fees. Sanctum stated that this will be a hot wallet to facilitate frequent fee adjustments, and that it may move to a bot hot wallet in the future. They stated that they are also considering the feasibility of deploying a pricing program that adjusts the fees algorithmically.

The SOL value calculator programs each also only have two authorities: The upgrade authority, and the manager.

The **calculator upgrade authority** can again replace the entire contract and should thus be heavily secured. A malicious upgrade obviously enables the attacker to steal all funds from the Infinity protocol. Sanctum has stated that these upgrade authorities will remain with the Sanctum multisig.

The **calculator manager** has very little power. They can only synchronize the last slot in which the external liquid staking protocol was upgraded. As such, they would have to collude with the upgrade authority of the external program to be able to DOS the protocol or steal funds. Sanctum has stated that it will be in a hot wallet during setup, and may move to a hardware wallet afterward.

External Authorities

Apart from the internal authorities of the Infinity protocol, external authorities can also affect it. Any authority of an external liquid staking program that can, for example, affect the data that the SOL value calculator programs access, that can affect the market price of the LST, that can pause the protocol, or that can trigger any similar actions, also has the power to affect the Infinity protocol.

The SOL value calculator programs already have a mechanism which pauses the calculator if the external program was upgraded and the calculator manager has not yet approved the new upgrade. This also means that any Infinity protocol trades or liquidity additions/removals with the affected LST are paused. Hence, **external upgrade authority compromises** are not, in themselves, a major concern as long as upgrades of the external programs are appropriately reviewed by the calculator manager.

Of greater concern are authorities that can change the data that the calculator program accesses. See the section on the on-chain data used by the calculator programs to understand what data is read.

In the case of **Marinade**, this is chiefly the **admin** authority, as it can update the stake account withdrawal fee, which is used in the intrinsic value calculation. However, that is hardcoded to be at most 0.2% in the marinade program, hence any changes here would have negligible effect. As such, we see the risk of such an attack as very low. Indeed, Marinade's authority structure is very restricted and the accounts are, as far as we can tell, managed securely. See our Marinade audit report for details.

In the case of **Lido**, no non-upgrade authority can directly change the data that is accessed by the calculator. There is no fee on withdrawing stake accounts, hence the SOL value calculator only uses the total SOL balance under Lido's control along with the stSOL supply to calculate the pricing.

In the case of **SPL Stake Pool**-managed LSTs, only the respective stake pool's **manager** can set the stake account withdrawal fee, but such changes are subject to the restriction that fee changes take two epoch boundaries to activate, and that the fees can be raised to at most 150% of the currently active fee. How this manager authority is held and secured depends entirely on the stake pool, but one would hope that most major LSTs using the stake pool program have this authority in a multisig, or at least a cold wallet.

Finally, some authorities can indirectly affect the market price of LSTs via, for example, setting the fee share of staking rewards or manipulating liquidity of the LST by (dis)incentivizing liquidity providers via fee changes. Much like it was discussed in the section on indirect smart contract risk and pool balance, this may represent a risk for Infinity protocol liquidity providers. The exact ways in which this can be done and in which it can affect the Infinity protocol are hard to predict. We recommend close monitoring of the market situation.

Synchronization Issues

What are Synchronization Issues?

Liquid staking protocols need to keep track of rewards their staking accounts receive at epoch boundaries, and adjust their LST valuation accordingly. This is usually done via a cranker instruction, often in conjunction with a project-operated cranker bot that continually calls that instruction after epoch boundaries. The protocols need to mitigate several attack vectors that arise from this – for example, they need to make it unprofitable for users to steal rewards by swapping into the LST before the update, then out of the LST afterward.

Similar attack vectors must be mitigated for the Infinity protocol, with the added difficulty there are now **two update processes**: After the epoch boundary, the liquid staking protocols must first update

their intrinsic valuation, after which the Infinity protocol must then synchronize the SOL value of its pool assets with the new intrinsic value. In this section, we discuss implications of this.

The updating of the SOL valuation of the pools works via the `SyncSolValue` instruction. Sanctum has stated that, like most liquid staking protocols, they also intend to maintain a cranker bot to operate this instruction in a timely manner.

Choosing Fees to Avoid Economic Attacks

Let's look at an epoch boundary at time E. For each LST i, let us define the times P_i and R_i as the time that the external protocol updates its internal LST price, and the time that the Infinity protocol updates the SOL value of its pool reserves for that LST, respectively. We have $E < P_i < R_i$. (Note that for some protocols like Marinade, which has to iterate over a potentially large collection of stake accounts, P_i may not be a distinct point in time. However, without loss of generality, we will treat it as one here.)

Unprivileged attackers can only interact with the pool by adding or removing liquidity, and by swapping between LSTs.

The LST prices used by Sanctum when **swapping** from LST i to j change at time P_i and P_j. Since the SOL value of the LSTs increase, ignoring fees, they would make about one additional epoch of rewards worth of profit if $P_i < P_j$ and they swap in the time interval (P_i, P_j). To mitigate this, Sanctum swap fees must be chosen such that for any pair of LSTs, the fees cover at least the one epoch of price increase. It may be prudent to set them slightly higher to adjust for varying rewards, DOS of one of the external protocols, network DOS or similar.

The price used when **adding and removing liquidity** via LST i changes at time P_i, and at times R_k for all LSTs k. Note that a liquidity operation automatically syncs the value of the underlying LST i, so for liquidity operations we can treat $P_i = R_i$.

The most profit an attacker can gain when adding liquidity via LST i is when they swap after P_i and before R_k for all $k \neq i$. When removing liquidity for LST j, the optimal configuration happens if they swap after R_k for all $k \neq j$ but before P_j. In both operations, they gain about one additional epoch of rewards. Note that both of these can be triggered in one epoch: Ignoring fees, if we have $P_i = R_i < R_k < R_j = P_j$ for all $k \neq i, j$ and the attacker adds liquidity via LST i directly after P_i and removes it again via LST j directly before P_j, they make approximately two additional epochs worth of rewards in profit. Hence, liquidity fees must be chosen such that they counteract at least two epochs worth of rewards in total.

Rebalance Authority is Exempt from Fee-Based Economic Attack Mitigations

There is one special case for fee-based mitigations to economic attacks as described above, which is the rebalance authority. During the rebalance operation, it can withdraw an arbitrary amount of one LST and deposit any amount of any other LSTs, and the rebalance operation will succeed as long as the total SOL value of the pool did not decrease. In particular, this means it can effectively swap between LSTs without being subject to fees.

The obvious consequence is that the fee-based mitigations do not work against economic attacks by the rebalance authority. As long as one of the price updates P_i happens during the rebalance operation, it can make a profit by abusing the price change. For example, it can start a rebalance operation, withdraw a small amount of LST i, trigger price change P_j for $j \neq i$ by calling the update cranker instruction of LST j, then end the rebalance. As long as the price change of LST j covers the amount of LST i that the rebalance authority withdrew, this succeeds.

The rebalance authority being exempt from fees is integral, since this allows the pool to rebalance without requiring large subsidies. As such, removing the fee exemption does not seem feasible.

Sanctum has stated that the rebalance authority will remain with them. Together with the fact that the potential impact of a compromise is, in all likelihood, only one epoch of rewards, we do not see this as a major risk factor.

Lido's Cranker

One final thing of note is that the state of Lido's cranker, which again is responsible for updating the protocol with the rewards that were received, is unclear. Lido on Solana has shut down its operations, and is to the best of our knowledge no longer being maintained. In order for Sanctum to support both swaps and liquidity operations with Lido's stSOL, the cranker must run at the start of every epoch. It is unclear if Lido will continue operating their cranker.

It may be necessary for Sanctum to run a separate Lido cranker if Lido's cranker ceases to operate and they wish to continue supporting Lido swaps. However, since Lido's cranker is open-source, this would only require minimal setup.

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

Neodyme assesses the severity of findings using the following general vulnerability classification system. Note that bugs are considered on a case-by-case basis and that we may deviate from these guidelines in exceptional circumstances.

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