Security Assessment Report

xORCA Staking Program

September 24, 2025

Summary

The Sec3 team was engaged to conduct a thorough security analysis of the xORCA Staking Program.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

The initial audit focused on the following versions and revealed 9 issues or questions.

Task	Type	Commit
xORCA Staking Program	Solana	8affaea601723bb9a2ed815c9f6e5586f68f7b55

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview 3
Findings in Detail 4
[H-01] Potential xorca vault inflation attack risk 4
[L-01] Potential pre-created PDA and ATA account risk 6
[L-02] Incorrect state account validation 8
[L-03] Missing validation of `INITIAL_UPGRADE_AUTHORITY_ID` 10
[L-04] Potential DoS if `PendingWithdraw` account has lamports before initialization 11
[I-01] Unnecessary writable modifier 12
[I-02] Incorrect event parameter 13
[I-03] Unsafe account closure procedure 14
[I-04] Optimize `AccountDiscriminator` 15
Appendix: Methodology and Scope of Work 16

Result Overview

Issue	Impact	Status
XORCA STAKING PROGRAM		
[H-01] Potential xorca vault inflation attack risk	High	Resolved
[L-01] Potential pre-created PDA and ATA account risk	Low	Acknowledged
[L-02] Incorrect state account validation	Low	Resolved
[L-03] Missing validation of `INITIAL_UPGRADE_AUTHORITY_ID`	Low	Resolved
[L-04] Potential DoS if `PendingWithdraw` account has lamports before initialization	Low	Resolved
[I-01] Unnecessary writable modifier	Info	Resolved
[I-02] Incorrect event parameter	Info	Resolved
[I-03] Unsafe account closure procedure	Info	Resolved
[I-04] Optimize `AccountDiscriminator`	Info	Resolved

Findings in Detail

XORCA STAKING PROGRAM

[H-01] Potential xorca vault inflation attack risk

Identified in commit `8affaea`.

The protocol allows users to stake ORCA tokens in order to mint XORCA tokens, where the XORCA token exchange rate is calculated as `non_escrowed_orca_amount / xorca_supply`.

```
/* solana-program/src/util/math.rs */
004 | pub fn convert_orca_to_xorca(
005 |     orca_amount_to_convert: u64,
006 |     non_escrowed_orca_amount: u64,
007 |     xorca_supply: u64,
008 | ) -> Result<u64, ProgramError> {
009 |     if (xorca_supply == 0) || (non_escrowed_orca_amount == 0) {
010 |         return Ok(orca_amount_to_convert);
011 |     }
013 |     // Perform calculations using u128 to prevent overflow for intermediate products.
014 |     // Convert all relevant u64 inputs to u128 for the calculation.
015 |     let xorca_supply_u128 = xorca_supply as u128;
016 |     let orca_amount_to_convert_u128 = orca_amount_to_convert as u128;
017 |     let non_escrowed_orca_amount_u128 = non_escrowed_orca_amount as u128;
019 |     let out_xorca_amount_u128 = orca_amount_to_convert_u128
020 |         .checked_mul(xorca_supply_u128)
021 |         .ok_or(ErrorCode::ArithmeticError)?
022 |         .checked_div(non_escrowed_orca_amount_u128)
023 |         .ok_or(ErrorCode::ArithmeticError)?;
025 |     // Cast the final u128 result back to u64.
026 |     let out_xorca_amount: u64 = out_xorca_amount_u128
027 |         .try_into()
028 |         .map_err(|_| ErrorCode::ArithmeticError)?; // Return an error if the value is too large for u64
030 |     Ok(out_xorca_amount)
031 | }
```

In particular,

- `non_escrowed_orca_amount` = protocol ORCA vault ATA balance - `escrowed_orca_amount`
- `escrowed_orca_amount` = ORCA token amount that have already been requested for withdrawal
- `xorca_supply` = the circulating supply of XORCA tokens

```
/* solana-program/src/instructions/stake.rs */
081 | let vault_account_data =
082 |     make_owner_token_account_assertions(vault_account, state_account, orca_mint_account)?;
084 | // Calculate xOrca to mint
085 | // Use checked math to guard against vault < escrow (should not happen, but defensive)
086 | let non_escrowed_orca_amount = vault_account_data
087 |     .amount
```

```
088 |     .checked_sub(state.escrowed_orca_amount)
089 |     .ok_or(ErrorCode::InsufficientVaultBacking)?;
091 | let xorca_to_mint = convert_orca_to_xorca(
092 |     *orca_stake_amount,
093 |     non_escrowed_orca_amount,
094 |     xorca_mint_data.supply,
095 | )?;
097 | if xorca_to_mint == 0 {
098 |     return Err(ErrorCode::InsufficientStakeAmount.into());
099 | }
```

In other words, the formula can be expressed as `(vault ATA account balance - escrowed_orca_amount) / xorca_supply`.

As a result, a malicious user can donate and directly transfer ORCA tokens into the vault ATA account to manipulate the XORCA token exchange rate. This creates a classic vault inflation attack scenario.

Consider the following example:

- The first staking user deposits 1 unit of ORCA, minting 1 XORCA (exchange rate = 1).
- The same user then transfers 1000e6 units of ORCA directly to the vault ATA account (exchange rate = 1000e6).

After this point, any staking attempt with an amount less than 1000e6 ORCA will fail with an `InsufficientStakeAmount` error.

Even though some large stakings are successful, the value of the minted tokens can be much lower than the value of the staked tokens. As a result, the malicious user can benefit from the rounding errors and steal from other stakers.

To mitigate this issue, it is recommended to defend with a virtual offset or permanently freeze a "dead share" during the first staking process. For example, enforce the minting of `100` or `1000` XORCA to a protocol-owned PDA address when the first staker joins.

Resolution

Fixed by commits `a6bf7cb` and `7ab1114`.

XORCA STAKING PROGRAM

[L-01] Potential pre-created PDA and ATA account risk

Identified in commit `8affaea`.

In the `initialize` instruction, the protocol creates and initializes both the `state` PDA account and the `vault` ATA account.

```
/* solana-program/src/instructions/initialize.rs */
126 | create_program_account_borsh(
127 |     system_program_account,
128 |     payer_account,
129 |     state_account,
130 |     &[state_seeds.as_slice().into()],
131 |     &state_data,
132 | )?;
134 | // Create the vault ATA using CPI
135 | let create_ata_ix = pinocchio::instruction::Instruction {
136 |     program_id: &ASSOCIATED_TOKEN_PROGRAM_ID,
137 |     accounts: &[
138 |         pinocchio::instruction::AccountMeta::writable_signer(payer_account.key()),
139 |         pinocchio::instruction::AccountMeta::writable(vault_account.key()),
140 |         pinocchio::instruction::AccountMeta::readonly(state_account.key()),
141 |         pinocchio::instruction::AccountMeta::readonly(orca_mint_account.key()),
142 |         pinocchio::instruction::AccountMeta::readonly(system_program_account.key()),
143 |         pinocchio::instruction::AccountMeta::readonly(token_program_account.key()),
144 |         pinocchio::instruction::AccountMeta::readonly(&ASSOCIATED_TOKEN_PROGRAM_ID),
145 |     ],
146 |     data: &[],
147 | };
148 | pinocchio::program::invoke(
149 |     &create_ata_ix,
150 |     &[
151 |         payer_account,
152 |         vault_account,
153 |         state_account,
154 |         orca_mint_account,
155 |         system_program_account,
156 |         token_program_account,
157 |         associated_token_program_account,
158 |     ],
159 | )?;
```

However, since the seeds for the `state` PDA account and `vault` ATA account are deterministic, once the program is deployed, any user can predict the `state` PDA account address and pre-create the `vault` ATA account.

```
/* solana-program/src/instructions/initialize.rs */
035 | let mut state_seeds = State::seeds();
036 | let state_bump = assert_account_seeds(state_account, &crate::ID, &state_seeds)?;
108 | // Verify vault address using centralized seeds
109 | let vault_seeds: Vec<Seed> = crate::pda::seeds::vault_seeds(
```

```
110 |     state_account.key(),
111 |     &SPL_TOKEN_PROGRAM_ID,
112 |     orca_mint_account.key(),
113 | );
114 | assert_account_seeds(vault_account, &ASSOCIATED_TOKEN_PROGRAM_ID, &vault_seeds)?;
```

A malicious user could exploit this weakness and break the initialization process by:

- transferring SOL to the state PDA account address,
- or pre-creating the vault ATA account

It is recommended that the protocol owner promptly invoke the `initialize` instruction immediately after program deployment.

Resolution

The team acknowledged this finding.

XORCA STAKING PROGRAM

[L-02] Incorrect state account validation

Identified in commit `8affaea`.

In the unstake instruction, before reading the `escrowed_orca_amount` value from the `state_account`, the program first validates whether the `state_account` is legitimate.

```
/* solana-program/src/instructions/unstake.rs */
102 | // Calculate withdrawable ORCA amount using checked math
103 | let initial_escrowed_orca_amount = {
104 |     let state_view = assert_account_data::<State>(state_account)?;
105 |
106 |     // Verify vault address using stored vault_bump
107 |     State::verify_vault_address_with_bump(
108 |         state_account,
109 |         vault_account,
110 |         orca_mint_account,
111 |         state_view.vault_bump,
112 |     )
113 |     .map_err(|_| ErrorCode::InvalidSeeds)?;
114 |
115 |     state_view.escrowed_orca_amount
116 | };
```

However, the validation logic is incorrectly implemented — it attempts to validate the `vault` account instead of the `state` account. The `vault` account had already been validated earlier.

```
/* solana-program/src/instructions/unstake.rs */
056 | // 3. Vault Account Assertions
057 | // Use stored vault_bump for verification - more efficient than assert_account_seeds
058 | let vault_account_data =
059 |     make_owner_token_account_assertions(vault_account, state_account, orca_mint_account)?;
```

Since `assert_account_data` validates the account discriminator, and the `state` account owner has already been validated, this does not pose a security risk.

```
/* solana-program/src/assertions/account.rs */
130 | pub fn assert_account_data<T: ProgramAccount>(
131 |     account: &AccountInfo,
132 | ) -> Result<Ref<'_, T>, ProgramError> {
133 |     assert_account_len(account, T::LEN)?;
134 |     assert_account_discriminator(account, &[T::DISCRIMINATOR])?;
135 |
136 |     let data = account.try_borrow_data()?;
137 |     Ok(T::from_bytes(data))
138 | }

/* solana-program/src/instructions/unstake.rs */
050 | // 2. xOrca State Account Assertions
```

```
051 | assert_account_role(state_account, &[AccountRole::Writable])?;
052 | assert_account_owner(state_account, &crate::ID)?;
```

Consider adding the validation logic to verify the `state` account properly. For example,

```
State::verify_address_with_bump(state_account, &crate::ID, state_view.bump).map_err(|_| ErrorCode::InvalidSeeds)?;
```

Resolution

Fixed by commit `c18bd89`.

XORCA STAKING PROGRAM

[L-03] Missing validation of `INITIAL_UPGRADE_AUTHORITY_ID`

> *Identified in commit* `8affaea`.

In the `initialize` instruction, `state_data.update_authority` is set to the `update_authority_account`, which is the `INITIAL_UPGRADE_AUTHORITY_ID`.

```
/* solana-program/src/instructions/initialize.rs */
078 | // 5. Update Authority Account Assertions
079 | assert_account_address(update_authority_account, &INITIAL_UPGRADE_AUTHORITY_ID)?;

124 | state_data.update_authority = *update_authority_account.key();
```

The update authority account can modify protocol settings through the `set` instruction, which requires the `update_authority_account` to be both `signer` and `writable`.

```
/* solana-program/src/instructions/set.rs */
020 | // 1. Update Authority Account Assertions
021 | assert_account_role(
022 |     update_authority_account,
023 |     &[AccountRole::Signer, AccountRole::Writable],
024 | )?;
```

It is recommended to add explicit `signer` and `writable` checks for the `INITIAL_UPGRADE_AUTHORITY_ID`.

Resolution

Fixed by commit `dfe2152`.

XORCA STAKING PROGRAM

[L-04] Potential DoS if `PendingWithdraw` account has lamports before initialization

Identified in commit `8affaea`.

Throughout the program, all account creation operations utilize the `create_account` function, which directly invokes the system program's `CreateAccount` instruction via CPI.

```
/* solana-program/src/util/account.rs */
029 | pub fn create_account(
036 | ) -> ProgramResult {
037 |     if new_account.is_owned_by(&SYSTEM_PROGRAM_ID) {
038 |         let rent = Rent::get()?;
039 |         let lamports = rent.minimum_balance(space);
041 |         CreateAccount {
042 |             program: system_program,
043 |             from: funder,
044 |             to: new_account,
045 |             lamports,
046 |             space: space as u64,
047 |             owner,
048 |         }
049 |         .invoke_signed(signers)?;
050 |     }
052 |     Ok(())
053 | }
```

The `system_instruction::create_account` method creates new accounts and will fail if the account's lamports are non-zero.

The `PendingWithdraw` account's PDA is generated using the seeds `["pending_withdraw", unstaker, withdraw_index]`, where `withdraw_index` is of type `u8`. If the account address of the unstaker is known, an attacker can transfer a small amount of SOL within the range of 0-255 to prevent the unstaker from creating any `PendingWithdraw` account, effectively blocking the unstake operation.

It's recommended to implement Anchor's initialization logic: When account lamports are not zero, use a combination of `transfer + allocate + assign` for account creation. This strategy helps prevent potential DoS attacks where malicious actors might pre-fund target accounts with lamports before creation attempts.

Resolution

Fixed by commit `b78f3c4`.

XORCA STAKING PROGRAM

[I-01] Unnecessary writable modifier

Identified in commit `8affaea`.

In the `Unstake` instruction, the `vault_account` is marked as writable.

```
/* solana-program/src/instructions/mod.rs */
034 | #[account(0, writable, signer, name = "unstaker_account")]
035 | #[account(1, writable, name = "state_account")]
036 | #[account(2, writable, name = "vault_account")]
037 | #[account(3, writable, name = "pending_withdraw_account")]
038 | #[account(4, writable, name = "unstaker_xorca_ata")]
039 | #[account(5, writable, name = "xorca_mint_account")]
040 | #[account(6, name = "orca_mint_account")]
041 | #[account(7, name = "system_program_account")]
042 | #[account(8, name = "token_program_account")]
043 | Unstake {
044 |     xorca_unstake_amount: u64,
045 |     withdraw_index: u8,
046 | },
```

However, during the execution of the `Unstake` instruction, the `vault_account` is never modified. Therefore, the writable modifier is unnecessary. The `vault_account` writable check in the `make_owner_token_account_assertions` function is also redundant.

```
/* solana-program/src/assertions/account.rs */
197 | pub fn make_owner_token_account_assertions<'a>(
198 |     owner_token_account: &'a AccountInfo,
199 |     owner_account: &AccountInfo,
200 |     token_mint_account: &AccountInfo,
201 | ) -> Result<TokenAccount, ProgramError> {
202 |     assert_account_role(owner_token_account, &[AccountRole::Writable])?;

/* solana-program/src/instructions/unstake.rs */
056 | // 3. Vault Account Assertions
057 | // Use stored vault_bump for verification - more efficient than assert_account_seeds
058 | let vault_account_data =
059 |     make_owner_token_account_assertions(vault_account, state_account, orca_mint_account)?;
```

Resolution

Fixed by commit `b949ad2`.

XORCA STAKING PROGRAM

[I-02] Incorrect event parameter

Identified in commit `8affaea`.

In the `Unstake` event, the `vault_xorca_amount` parameter should be `vault_orca_amount`.

```
/* solana-program/src/event.rs */
015 | Unstake {
016 |     xorca_unstake_amount: &'a u64,
017 |     vault_xorca_amount: &'a u64,
018 |     vault_escrowed_orca_amount: &'a u64,
019 |     xorca_mint_supply: &'a u64,
020 |     withdrawable_orca_amount: &'a u64,
021 |     cool_down_period_s: &'a i64,
022 |     withdraw_index: &'a u8,
023 | },
```

In `Stake`, `Unstake`, and `Withdraw`, events like `vault_orca_amount` and `xorca_mint_supply` reflect the outdated state from before the instruction was executed.

```
/* solana-program/src/instructions/stake.rs */
101 | // Transfer Orca from staker ATA to vault
102 | let transfer_instruction = Transfer {
103 |     from: staker_orca_ata,
104 |     to: vault_account,
105 |     authority: staker_account,
106 |     amount: *orca_stake_amount,
107 | };
108 | transfer_instruction.invoke()?;
110 | // Mint xOrca to staker xOrca ATA
111 | let mint_to_instruction = MintTo {
112 |     mint: xorca_mint_account,
113 |     account: staker_xorca_ata,
114 |     mint_authority: state_account,
115 |     amount: xorca_to_mint,
116 | };
117 | mint_to_instruction.invoke_signed(&[state_seeds.as_slice().into()])?;
119 | Event::Stake {
120 |     orca_stake_amount: orca_stake_amount,
121 |     vault_orca_amount: &vault_account_data.amount,
122 |     vault_escrowed_orca_amount: &state.escrowed_orca_amount,
123 |     xorca_mint_supply: &xorca_mint_data.supply,
124 |     xorca_to_mint: &xorca_to_mint,
125 | }
126 | .emit()?;
```

Resolution

Fixed by commit `b949ad2`.

XORCA STAKING PROGRAM

[I-03] Unsafe account closure procedure

Identified in commit `8affaea`.

In the current implementation of the program, when closing an account, the `close_program_account` function performs the following actions:

1. Sets the discriminator of `account_to_close` to `AccountDiscriminator::Closed`.
2. Transfers the lamports from `account_to_close` to the `receiver`.
3. Sets the lamports of `account_to_close` to `0`.

```
/* solana-program/src/util/account.rs */
104 | pub fn close_program_account(
105 |     account_to_close: &AccountInfo,
106 |     receiver: &AccountInfo,
107 | ) -> ProgramResult {
108 |     let mut account_to_close_data = account_to_close.try_borrow_mut_data()?;
109 |     account_to_close_data[0] = AccountDiscriminator::Closed as u8;
110 |     *receiver.try_borrow_mut_lamports()? += account_to_close.lamports();
111 |     *account_to_close.try_borrow_mut_lamports()? = 0;
112 |     Ok(())
113 | }
```

Although it's safe in the current implementation because of the discriminator checks, it's still recommended to assign the account owner to the system program and reallocate the account size to `0`.

Resolution

Fixed by commit `24d21bd`.

XORCA STAKING PROGRAM

[I-04] Optimize `AccountDiscriminator`

> *Identified in commit* `8affaea`.

The current implementation of the program defines a custom `AccountDiscriminator` enum, where the default value represents the `State` account.

```
/* solana-program/src/state/mod.rs */
012 | #[repr(u8)]
013 | pub enum AccountDiscriminator {
014 |     State,           // 0
015 |     PendingWithdraw, // 1
016 |     Closed,          // 2
017 | }
```

It would be better to set the first enum variant (value `0`) to `Uninitialized` to cover all cases of uninitialized accounts.

However, since the `State` Account in this program is a global PDA, the current implementation is considered safe.

Resolution

Fixed by commit `b949ad2`.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Orca Management Company, S.A (the "Client''). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.