Launchpad (Solana)

Smart Contract Audit Report
Prepared for DAgora

Date Issued:	Aug 21, 2023
Project ID:	AUDIT2022052
Version:	v1.0
Confidentiality Level:	Public

inspex

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022052
Version	v1.0
Client	DAgora
Project	Launchpad (Solana)
Auditor(s)	Peeraphut Punsuwan Puttimet Thammasaeng Ronnachai Chaipha
Author(s)	Ronnachai Chaipha
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Aug 21, 2023	Full report	Ronnachai Chaipha

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by DAgora, Inspex team conducted an audit to verify the security posture of the Launchpad (Solana) smart contracts between Nov 1, 2022 and Nov 3, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Launchpad (Solana) smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 high, 2 medium, 4 low-severity issues. With the project team's prompt response, 2 high, 2 medium, 2 low-severity issues were resolved or mitigated in the reassessment, while 2 low-severity issues were acknowledged by the team. Therefore, Inspex trusts that Launchpad (Solana) smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

This smart contract passes Inspex's security verification standard, and is trustworthy.

Approved by Inspex on Aug 21, 2023

inspex CYBERSECURITY PROFESSIONAL SERVICE

PASS

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

DAgora Launchpad is a project that allows the user who wants to have their launchpad contracts created to do so on their own. These launchpads are used to offer NFT redemption to their platforms' users. It also includes all necessary functions to support the business design for the launchpad creators. In exchange, there will be a fee collected for the DAgora.

Scope Information:

Project Name	Launchpad (Solana)
Website	https://dagora.xyz/
Smart Contract Type	Solana Program
Chain	Solana
Programming Language	Rust
Category	NFT, Launchpad

Audit Information:

Audit Method	Whitebox
Audit Date	Nov 1, 2022 - Nov 3, 2022
Reassessment Date	Nov 25, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit

Contract	Bytecode SHA256 Hash
dagora_launchpad	d34565f92018ef42811bbb3913ee3b4aa4b063a02bf05bba14b00892f55cc1c4

Reassessment

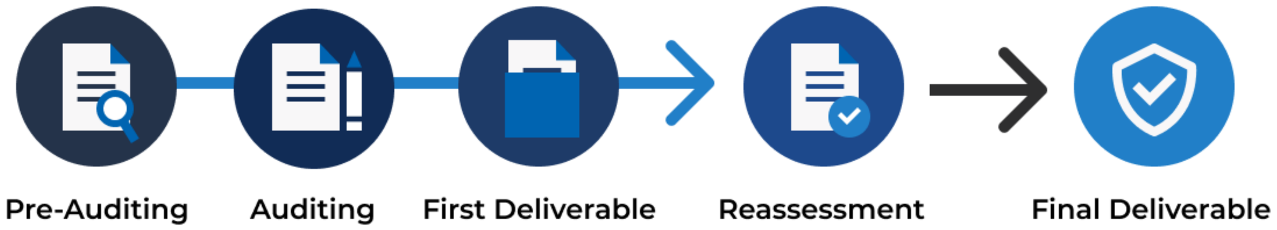
Contract	Bytecode SHA256 Hash
dagora_launchpad	ce3dced175524c4c1730fc65b2148528c7aa8e3e7b96f28a91503319b6d44de3

As the DAgora team has decided not to publish the source code to protect their intellectual property, the users should compare the bytecode hashes with the smart contracts before interacting with them to make sure that they are the same with the contracts audited.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

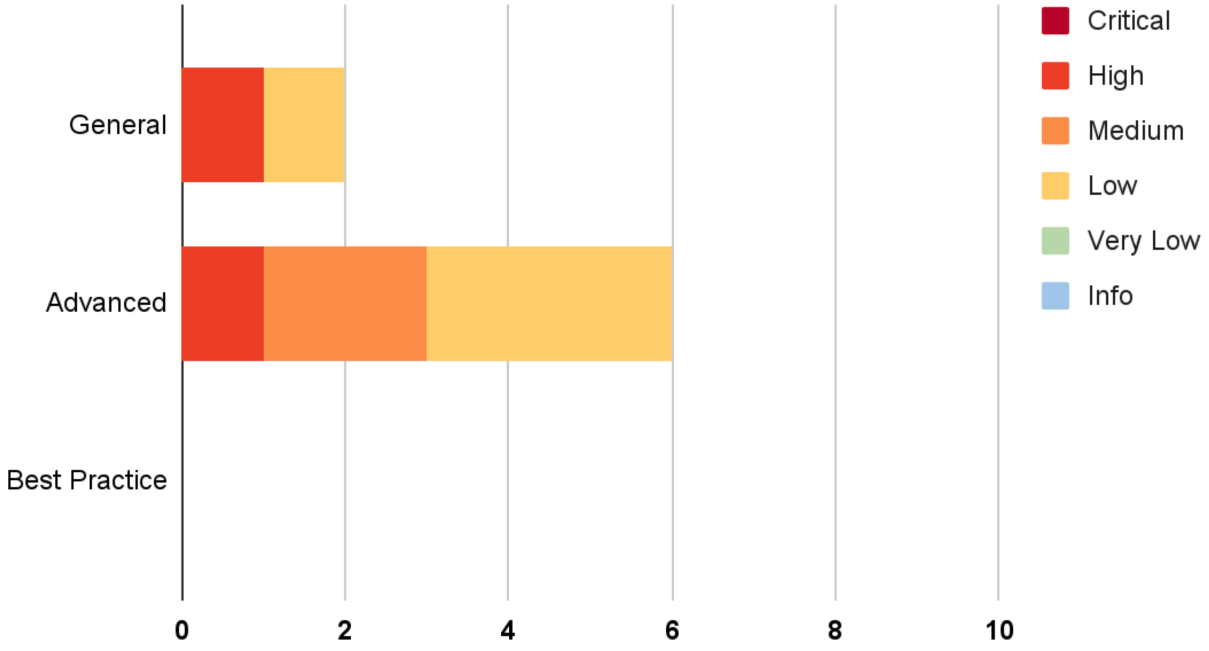
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

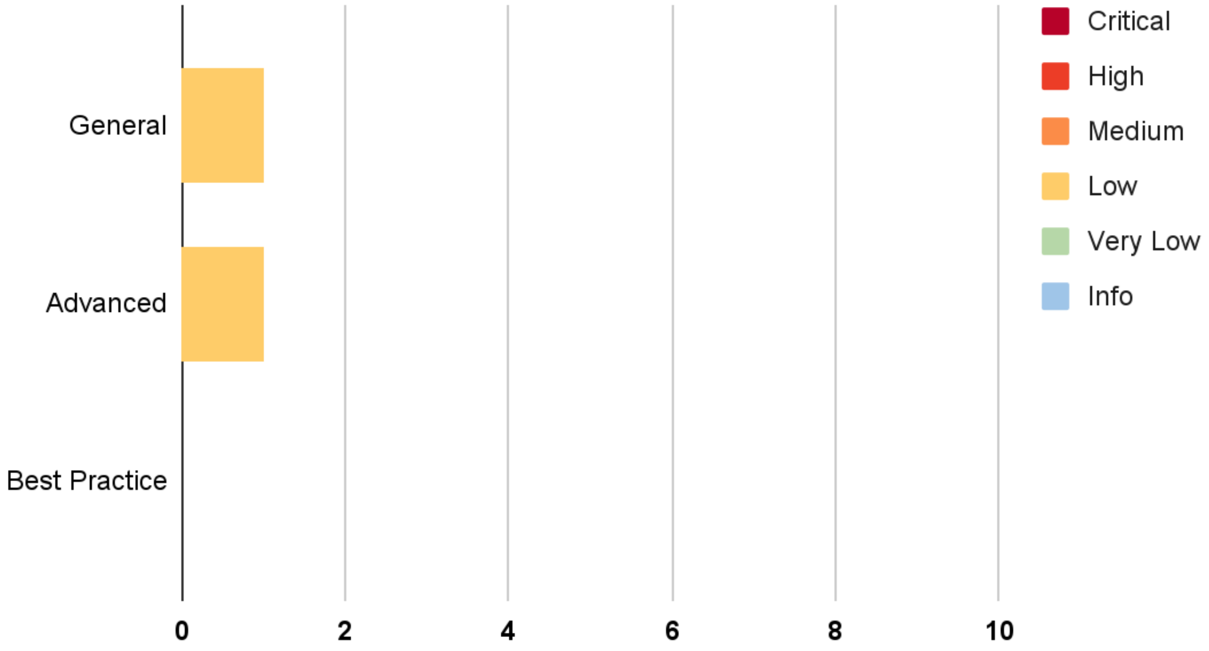
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Upgradability of Solana Program	General	High	Resolved *
IDX-002	Account Collision	Advanced	High	Resolved
IDX-003	Improper Token Withdrawal from Launchpads	Advanced	Medium	Resolved
IDX-004	Improper Launchpad Setting	Advanced	Medium	Resolved
IDX-005	Missing of Launchpad Register Validation	Advanced	Low	Resolved
IDX-006	Insecure Source of Randomness	Advanced	Low	Acknowledged
IDX-007	Improper Fee Enforcement	Advanced	Low	Resolved
IDX-008	Smart Contract with Unpublished Source Code	General	Low	Acknowledged

* The mitigations or clarifications by DAgora can be found in Chapter 5.

5. Detailed Findings Information

5.1. Upgradability of Solana Program

ID	IDX-001
Target	dagora_launchpad
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The logic of the affected programs can be arbitrarily changed. This allows the upgrade authority to change the logic of the program in favor to the platform, e.g., transferring the users' funds to the platform owner's account. **Likelihood: Medium** Only the program upgrade authority can redeploy the program to the same program address; however, there is no restriction to prevent the authority from inserting malicious logic.
Status	Resolved * The DAgora team has mitigated this issue by confirming that the upgrade authority will be a multisig account controlled by multiple trusted parties.

5.1.1. Description

Programs on Solana can be deployed through the upgradable BPF loader to make them upgradable, allowing the program's upgrade authority to redeploy the program with the new logic, bug fixes, or upgrades to the same program address.

However, there is no restriction on how and when the program will be upgraded. This opens up an attack surface on the program, allowing the upgrade authority to redeploy the program with malicious logic and gain unfair benefits from the users, for example, transferring funds out from the users' accounts.

5.1.2. Remediation

Inspex suggests deploying the program as an immutable program to prevent the program logic from being modified.

However, if the upgradability is needed, Inspex suggests mitigating this issue by the following options:

- Using a multisig account controlled by multiple trusted parties as the upgrade authority
- Implementing a community-run governance to control the redeployment of the program

5.2. Account Collision

ID	IDX-002
Target	dagora_launchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The whitelisted users will be unable to register on the mintable launchpad. **Likelihood:** Medium It is likely to happen. Since the attacker can use the users' address as a seed when claiming the NFT from a mintable launchpad to prevent the whitelisted users from the registration process in the launchpad.
Status	Resolved The DAgora team has resolved this issue as suggested by adding the seed constant for each account type.

5.2.1. Description

The `claim_mintable_launchpad()` function can be used to claim the NFT by passing the `nft_mint` account derived by the `seeds` parameter at line 339.

contexts.rs

```
289 #[derive(Accounts)]
290 #[instruction(seeds: Vec<u8>)]
291 pub struct ClaimMintableLaunchpadContext<'info> {
292   #[account(mut)]
293   pub user: Signer<'info>,
294
295   pub mintable_launchpad: Account<'info, MintableLaunchpad>,
296
297   /// CHECK: skip
298   #[account(
299     mut,
300     address = get_launchpad_authority(mintable_launchpad.key()).0
301   )]
302   pub launchpad_authority: AccountInfo<'info>,
303
304   #[account(
305     mut,
306     seeds = [
307         user.to_account_info().key().as_ref(),
308         mintable_launchpad.key().as_ref()
309       ],
310       bump = user_profile.nonce,
311     )]
312     pub user_profile: Account<'info, UserProfile>,
313
314     /// CHECK: checked by address
315     #[account(
316       mut,
317       address = mintable_launchpad.collection_mint
318     )]
319     pub collection_mint: AccountInfo<'info>,
320
321     /// CHECK: checked by address
322     #[account(
323       mut,
324       address = find_metadata_account(&collection_mint.key()).0
325     )]
326     pub collection_metadata: AccountInfo<'info>,
327
328     /// CHECK: checked by address
329     #[account(
330       mut,
331       address = find_master_edition_account(&collection_mint.key()).0
332     )]
333     pub collection_master_edition: AccountInfo<'info>,
334
335     /// CHECK: checked by seeds
336     #[account(
337       mut,
338       seeds = [
339         seeds.as_ref()
340       ],
341       bump
342     )]
343     pub nft_mint: AccountInfo<'info>,
344     ...
```

The `claim_mintable_launchpad()` function will create the NFT account in the `mintable_launchpad.claim_nft()` function and the `nft_mint` account will be passed as the `nft_mint_account` in line 370 as shown below:

lib.rs

```
347 pub fn claim_mintable_launchpad(
348     ctx: Context<ClaimMintableLaunchpadContext>,
    seeds: Vec<u8>,
) -> Result<()> {
    let user = &mut ctx.accounts.user;
    let mintable_launchpad = &ctx.accounts.mintable_launchpad;
    let launchpad_authority = &ctx.accounts.launchpad_authority;
    let user_profile = &mut ctx.accounts.user_profile;
    let collection_metadata = &ctx.accounts.collection_metadata;
    let collection_mint = &ctx.accounts.collection_mint;
    let collection_master_edition = &ctx.accounts.collection_master_edition;
    let nft_mint = &ctx.accounts.nft_mint;
    let receipt_account = &ctx.accounts.receipt_account;
    let token_metadata = &ctx.accounts.token_metadata;
    let ata_program = &ctx.accounts.ata_program;
    let token_program = &ctx.accounts.token_program;
    let system_program = &ctx.accounts.system_program;
    let sysvar_program = &ctx.accounts.sysvar_program;

    let launchpad_pubkey = &mintable_launchpad.to_account_info().key();

    user_profile.before_claim()?;

    mintable_launchpad.claim_nft(
      launchpad_pubkey,
      seeds,
      user,
      nft_mint,
      launchpad_authority,
      collection_metadata,
      collection_mint,
      collection_master_edition,
      receipt_account,
      token_metadata,
      ata_program,
      token_program,
      system_program,
      sysvar_program,
    )?;

    emit!(ClaimLaunchpadEvent {
      nft_mint: nft_mint.key()
    });

    Ok(())
}
```

In the `mintable_launchpad.claim_nft()` function, the `nft_mint_account` account is passed to the `create_account()` function as the `to_pubkey` parameter, as shown below in line 192.

states.rs

```
create_account(
 user,
 nft_mint_account,
 sysvar_program.minimum_balance(82),
 82,
 token_program,
 &[nft_signer_seeds],
)?;
```

The `create_account()` function is called to create an account for the NFT at the address of the `to_pubkey` account.

utils.rs

```
pub fn create_account<'info>(
 from_pubkey: &AccountInfo<'info>,
 to_pubkey: &AccountInfo<'info>,
 lamports: u64,
 space: u64,
 owner: &AccountInfo<'info>,
 signer_seeds: &[&[&[u8]]]
) -> Result<()> {
 let create_account_instruction = system_instruction::create_account(
   &from_pubkey.key(),
   &to_pubkey.key(),
   lamports,
   space,
   &owner.key(),
 );

 if signer_seeds.is_empty() {
  solana_program::program::invoke(
    &create_account_instruction,
    &[
      from_pubkey.clone(),
      to_pubkey.clone()
    ]
  )?;
 } else {
  solana_program::program::invoke_signed(
    &create_account_instruction,
    &[
      from_pubkey.clone(),
      to_pubkey.clone()
    ],
    signer_seeds
    )?;
  }

  Ok(())
}
```

After the `claim_mintable_launchpad()` function, the `nft_mint` address will be used for the minted NFT.

When the attacker uses the victim's address combined with the target launchpad's address as a `seeds` parameter of the `claim_mintable_launchpad()` function, the NFT account will be created at the same address as the `user_profile` account which created from the `create_user_profile()` function as shown below in lines 133-140.

contexts.rs

```
122 #[derive(Accounts)]
123 pub struct CreateUserProfileContext<'info> {
124   #[account(mut)]
125   pub user: Signer<'info>,
126 
127   /// CHECK: we check later
128   pub launchpad: AccountInfo<'info>,
129 
130   #[account(
131     init,
132     seeds = [
133       user.to_account_info().key().as_ref(),
134       launchpad.key().as_ref()
135     ],
136     bump,
137     payer = user,
138     space = 8 + UserProfile::LEN
139   )]
140   pub user_profile: Account<'info, UserProfile>,
141 
142   pub system_program: Program<'info, System>
143 }
```

Due to the account collision, the whitelisted users will be unable to register in both mintable and transferable launchpads.

5.2.2. Remediation

Inspex suggests including the seed's prefix in the PDA account creation in order to ensure that each account's address is unique from any other.

For example, adding the `b"USER_PROFILE"` prefix to the seed of the `user_profile` creation as shown in line 133.

contexts.rs

```
122 #[derive(Accounts)]
123 pub struct CreateUserProfileContext<'info> {
124   #[account(mut)]
125   pub user: Signer<'info>,
126 
127   /// CHECK: we check later
128   pub launchpad: AccountInfo<'info>,
129 
130   #[account(
131     init,
132     seeds = [
133       b"USER_PROFILE",
134       user.to_account_info().key().as_ref(),
135       launchpad.key().as_ref()
136     ],
137     bump,
138     payer = user,
139     space = 8 + UserProfile::LEN
140   )]
141   pub user_profile: Account<'info, UserProfile>,
142 
143   pub system_program: Program<'info, System>
144 }
```

Adding the `b"MINTABLE_LAUNCHPAD"` prefix to the seed of the `nft_mint` creation as shown in line 339.

contexts.rs

```
289 #[derive(Accounts)]
290 #[instruction(seeds: Vec<u8>)]
291 pub struct ClaimMintableLaunchpadContext<'info> {
292   #[account(mut)]
293   pub user: Signer<'info>,
294 
295   pub mintable_launchpad: Account<'info, MintableLaunchpad>,
296 
297   /// CHECK: skip
298   #[account(
299     mut,
300     address = get_launchpad_authority(mintable_launchpad.key()).0
301   )]
302   pub launchpad_authority: AccountInfo<'info>,
303 
304   #[account(
305     mut,
306     seeds = [
307       user.to_account_info().key().as_ref(),
    mintable_launchpad.key().as_ref()
  ],
  bump = user_profile.nonce,
)]
pub user_profile: Account<'info, UserProfile>,

/// CHECK: checked by address
#[account(
  mut,
  address = mintable_launchpad.collection_mint
)]
pub collection_mint: AccountInfo<'info>,

/// CHECK: checked by address
#[account(
  mut,
  address = find_metadata_account(&collection_mint.key()).0
)]
pub collection_metadata: AccountInfo<'info>,

/// CHECK: checked by address
#[account(
  mut,
  address = find_master_edition_account(&collection_mint.key()).0
)]
pub collection_master_edition: AccountInfo<'info>,

/// CHECK: checked by seeds
#[account(
  mut,
  seeds = [
    b"MINTABLE_LAUNCHPAD",
    seeds.as_ref()
  ],
  bump
)]
pub nft_mint: AccountInfo<'info>,

/// CHECK: skip
#[account(mut)]
pub receipt_account: AccountInfo<'info>,

/// CHECK: skip
#[account(
  mut,
  address = find_metadata_account(&nft_mint.key()).0
)]
```

Remember to add the seed prefix to every context that uses the `user_profile` account, for example, at line 156.

contexts.rs

```
145 #[derive(Accounts)]
146 pub struct RegisterMintableLaunchpadContext<'info> {
147   #[account(mut)]
148   pub user: Signer<'info>,
149 
150   #[account(mut)]
151   pub mintable_launchpad: Account<'info, MintableLaunchpad>,
152 
153   #[account(
154     mut,
155     seeds = [
156       b"USER_PROFILE",
157       user.to_account_info().key().as_ref(),
158       mintable_launchpad.to_account_info().key().as_ref()
159     ],
160     bump = user_profile.nonce,
161     constraint = !user_profile.is_registered @ErrorCode::Registered
162   )]
163   pub user_profile: Account<'info, UserProfile>,
164 }
```

5.3. Improper Token Withdrawal from Launchpads

ID	IDX-003
Target	dagora_launchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** High The launchpad's owner can withdraw the NFT from the launchpad at any time. This results in the users not being able to claim the NFT from the launchpad. **Likelihood:** Low There is nothing to prevent the withdrawal from being done; however, this action can only be done by the launchpad's owner, which is set by the platform's owner.
Status	Resolved The DAgora team has resolved this issue as suggested by adding a mechanism that only allows the launchpad's owner to be able to withdraw before or after redemption time.

5.3.1. Description

In the DAgora launchpad project, both the `withdraw_token_from_mintable_launchpad()` and the `withdraw_token_from_transferable_launchpad()` functions are used by the launchpad's owner to withdraw the leftover tokens from the launchpad.

lib.rs

```rust
pub fn withdraw_token_from_mintable_launchpad(
    ctx: Context<WithdrawTokenFromMintableLaunchpadContext>,
    _token_mint: Pubkey,
    amount: u64,
) -> Result<()> {
    let mintable_launchpad = &ctx.accounts.mintable_launchpad;
    let launchpad_authority = &ctx.accounts.launchpad_authority;
    let launchpad_token_account = &ctx.accounts.launchpad_token_account;
    let owner_token_account = &ctx.accounts.owner_token_account;

    let launchpad_pubkey = &mintable_launchpad.to_account_info().key();
    let (_, authority_nonce): (Pubkey, u8) =
get_launchpad_authority(launchpad_pubkey.clone());

    let seeds: &[&[_]] = &[launchpad_pubkey.as_ref(), &[authority_nonce]];

    transfer_token(
        launchpad_authority,
        launchpad_token_account,
        owner_token_account,
        amount,
        &[seeds],
    )?;
    Ok(())
}

pub fn withdraw_token_from_transferable_launchpad(
    ctx: Context<WithdrawTokenFromTransferableLaunchpadContext>,
    _nft_mint: Pubkey,
    amount: u64,
) -> Result<()> {
    let transferable_launchpad = &ctx.accounts.transferable_launchpad;
    let launchpad_authority = &ctx.accounts.launchpad_authority;
    let launchpad_token_account = &ctx.accounts.launchpad_token_account;
    let owner_token_account = &ctx.accounts.owner_token_account;

    let launchpad_pubkey = &transferable_launchpad.to_account_info().key();
    let (_, authority_nonce): (Pubkey, u8) =
get_launchpad_authority(launchpad_pubkey.clone());

    let seeds: &[&[_]] = &[launchpad_pubkey.as_ref(), &[authority_nonce]];

    transfer_token(
        launchpad_authority,
        launchpad_token_account,
        owner_token_account,
        amount,
        &[seeds],
    )?;
    Ok(())
}
```

However, there is currently no constraint to prevent the token withdrawal before users claim the token.

5.3.2. Remediation

Inspex suggest adding validation to prevent the launchpad's owner from withdrawing the tokens before the claiming duration has ended, For example as shown below in lines 513-519 and 544-550:

lib.rs

```
508 pub fn withdraw_token_from_mintable_launchpad(
509     ctx: Context<WithdrawTokenFromMintableLaunchpadContext>,
510     _token_mint: Pubkey,
511     amount: u64,
) -> Result<()> {
    let mintable_launchpad = &mut ctx.accounts.mintable_launchpad;
    let current_time = Clock::get().unwrap().unix_timestamp;
    let duration = 604800; // a week
    require!(
      current_time >= mintable_launchpad.redeem_end_timestamp.checked_add(duration).unwrap(),
      ErrorCode::InvalidWithdrawTime
    );
    let launchpad_authority = &ctx.accounts.launchpad_authority;
    let launchpad_token_account = &ctx.accounts.launchpad_token_account;
    let owner_token_account = &ctx.accounts.owner_token_account;

    let launchpad_pubkey = &mintable_launchpad.to_account_info().key();
    let (_, authority_nonce): (Pubkey, u8) = get_launchpad_authority(launchpad_pubkey.clone());

    let seeds: &[&[_]] = &[launchpad_pubkey.as_ref(), &[authority_nonce]];

    transfer_token(
      launchpad_authority,
      launchpad_token_account,
      owner_token_account,
      amount,
      &[seeds],
    )?;
    Ok(())
}

pub fn withdraw_token_from_transferable_launchpad(
    ctx: Context<WithdrawTokenFromTransferableLaunchpadContext>,
    _nft_mint: Pubkey,
    amount: u64,
) -> Result<()> {
    let transferable_launchpad = &mut ctx.accounts.transferable_launchpad;
    let current_time = Clock::get().unwrap().unix_timestamp;
    let duration = 604800; // a week
    require!(
      current_time >= transferable_launchpad.get_launchpad_core().redeem_end_timestamp.checked_add(duration).unwrap(),
      ErrorCode::InvalidWithdrawTime
    );
    let launchpad_authority = &ctx.accounts.launchpad_authority;
    let launchpad_token_account = &ctx.accounts.launchpad_token_account;
    let owner_token_account = &ctx.accounts.owner_token_account;

    let launchpad_pubkey = &transferable_launchpad.to_account_info().key();
    let (_, authority_nonce): (Pubkey, u8) =
get_launchpad_authority(launchpad_pubkey.clone());

    let seeds: &[&[_]] = &[launchpad_pubkey.as_ref(), &[authority_nonce]];

    transfer_token(
      launchpad_authority,
      launchpad_token_account,
      owner_token_account,
      amount,
      &[seeds],
    )?;
    Ok(())
}
```

5.4. Improper Launchpad Setting

ID	IDX-004
Target	dagora_launchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium When the duration of the launchpad begins, the launchpad's owner can set the value of the launchpad, such as a minting fee, the maximum number of registered users, whitelisting root, etc. This results in the unreliability of the platform. **Likelihood:** Medium It is likely that the launchpad's owner will set a new value during the launchpad duration, but the users may observe the setting transaction before buying the token.
Status	Resolved The DAgora team has resolved this issue as suggested by adding the mechanism that prevents the launchpad's owner from updating the launchpad setting after the registration process has started.

5.4.1. Description

The launchpad's state variables can be updated at any time by the launchpad's owner. When the launchpad's owner can update a critical state, that can cause impacts on the users, such as updating the fee mint, setting new whitelisting, or updating the maximum redeem. To be fair to the users, after the launchpad has been started, the launchpad's owner should not be able to update the launchpad's state. The following source code contains the `set_mintable_launchpad()` function logic:

lib.rs

```
pub fn set_mintable_launchpad(
  ctx: Context<SetMintableLaunchpadContext>,
  // launchpad core
  is_active: bool,
  whitelist_root: Option<[u8; 32]>,
  max_register: u64,
  max_redeem: u64,
  fee_mint: Pubkey,
  fee_redeem: u64,
  register_start_timestamp: i64,
  register_end_timestamp: i64,
  redeem_start_timestamp: i64,
  redeem_end_timestamp: i64,
  max_per_user: u64,
  sharing_fee: u64,
  protocol_fee: u64,
  // mintable launchpad
  collection_mint: Pubkey,
  name: String,
  symbol: String,
  uri: String,
  seller_fee_basis_points: u16,
  royalty_fee_for_owner: u8,
) -> Result<()> {
  let mintable_launchpad = &mut ctx.accounts.mintable_launchpad;

  let launchpad_core: LaunchpadCore = LaunchpadCore {
    is_active,
    whitelist_root,
    max_register,
    max_redeem,
    total_register: 0,
    total_redeem: 0,
    fee_mint,
    fee_redeem,
    sharing_fee,
    protocol_fee,
    max_per_user,
    register_start_timestamp,
    register_end_timestamp,
    redeem_start_timestamp,
    redeem_end_timestamp,
  };
  mintable_launchpad.launchpad_core = launchpad_core;
  mintable_launchpad.total_nft_redeem = 0;
  mintable_launchpad.collection_mint = collection_mint;
  mintable_launchpad.name = name.clone();
  mintable_launchpad.symbol = symbol.clone();
  mintable_launchpad.uri = uri.clone();
  mintable_launchpad.seller_fee_basis_points = seller_fee_basis_points;
  mintable_launchpad.royalty_fee_for_owner = royalty_fee_for_owner;

  emit!(SetMintableLaunchpadEvent {
    is_active,
    whitelist_root,
    fee_mint,
    fee_redeem,
    register_start_timestamp,
    register_end_timestamp,
    redeem_start_timestamp,
    redeem_end_timestamp,
    max_per_user,
    collection_mint,
    name,
    symbol,
    uri,
    seller_fee_basis_points,
    royalty_fee_for_owner
  });

  Ok(())
}
```

According to the above source code, the owner of the launchpad can modify all settings, including `fee_mint`, `fee_redeem`, `fee_redeem`, and `protocol_fee`, to immediately increase or decrease the fee paid by users.

Furthermore, after making changes, the `total_register`, `total_redeem`, and `total_nft_redeem` states will be set to 0, causing the counting state to be invalid when validating in the `validate_max_register()` function.

states.rs

```
pub fn validate_max_register(&mut self) -> Result<()> {
  if self.max_register > 0 {
    require!(self.total_register + 1 <= self.max_register, ErrorCode::ReachMaxRegister);
  }

  self.total_register = self.total_register.checked_add(1).unwrap();
  Ok(())
}

pub fn validate_max_redeem(&mut self, amount: u64) -> Result<()> {
  if self.max_redeem > 0 {
    require!(self.total_redeem.checked_add(amount).unwrap() <= self.max_redeem, ErrorCode::ReachMaxRedeem);
  }

  self.total_redeem = self.total_redeem.checked_add(amount).unwrap();
  Ok(())
}
```

It results in other users being able to register or redeem more number than expected.

5.4.2. Remediation

Inspex suggests adding the condition to prevent the launchpad's owner from modifying the state of the launchpad after the registration process is started. In case the launchpad's owner sets the `register_start_timestamp` state to `0` to start the launchpad immediately, the `total_register` state must be checked to ensure that no user has already registered, For example in lines 105-108 and 190-193:

errors.rs

```
71  #[msg("Dagora Launchpad: Cannot update launchpad data at this time.")]
72  CannotUpdateLaunchpad,
```

lib.rs

```
79  pub fn set_mintable_launchpad(
80    ctx: Context<SetMintableLaunchpadContext>,
81    // launchpad core
82    is_active: bool,
83    whitelist_root: Option<[u8; 32]>,
84    max_register: u64,
85    max_redeem: u64,
86    fee_mint: Pubkey,
87    fee_redeem: u64,
88    register_start_timestamp: i64,
89    register_end_timestamp: i64,
90    redeem_start_timestamp: i64,
91    redeem_end_timestamp: i64,
92    max_per_user: u64,
93    sharing_fee: u64,
94    protocol_fee: u64,
95    // mintable launchpad
96    collection_mint: Pubkey,
97    name: String,
98    symbol: String,
99    uri: String,
100   seller_fee_basis_points: u16,
101   royalty_fee_for_owner: u8,
102 ) -> Result<()> {
103   let mintable_launchpad = &mut ctx.accounts.mintable_launchpad;
104
105   let current_time = Clock::get().unwrap().unix_timestamp;
106   let launchpad_core = mintable_launchpad.get_launchpad_core();
107   require!(launchpad_core.total_register == 0,
    ErrorCode::CannotUpdateLaunchpad);
108   require!(launchpad_core.register_start_timestamp > current_time ||
    launchpad_core.register_start_timestamp == 0,
    ErrorCode::CannotUpdateLaunchpad);
109
```

```
  let launchpad_core: LaunchpadCore = LaunchpadCore {
    is_active,
    whitelist_root,
    max_register,
    max_redeem,
    total_register: 0,
    total_redeem: 0,
    fee_mint,
    fee_redeem,
    sharing_fee,
    protocol_fee,
    max_per_user,
    register_start_timestamp,
    register_end_timestamp,
    redeem_start_timestamp,
    redeem_end_timestamp,
  };
  mintable_launchpad.launchpad_core = launchpad_core;
  mintable_launchpad.total_nft_redeem = 0;
  mintable_launchpad.collection_mint = collection_mint;
  mintable_launchpad.name = name.clone();
  mintable_launchpad.symbol = symbol.clone();
  mintable_launchpad.uri = uri.clone();
  mintable_launchpad.seller_fee_basis_points = seller_fee_basis_points;
  mintable_launchpad.royalty_fee_for_owner = royalty_fee_for_owner;

  emit!(SetMintableLaunchpadEvent {
    is_active,
    whitelist_root,
    fee_mint,
    fee_redeem,
    register_start_timestamp,
    register_end_timestamp,
    redeem_start_timestamp,
    redeem_end_timestamp,
    max_per_user,
    collection_mint,
    name,
    symbol,
    uri,
    seller_fee_basis_points,
    royalty_fee_for_owner
  });

  Ok(())
}
```

lib.rs

```rust
pub fn set_transferable_launchpad(
  ctx: Context<SetTransferableLaunchpadContext>,
  // launchpad core
  is_active: bool,
  whitelist_root: Option<[u8; 32]>,
  max_register: u64,
  max_redeem: u64,
  fee_mint: Pubkey,
  fee_redeem: u64,
  register_start_timestamp: i64,
  register_end_timestamp: i64,
  redeem_start_timestamp: i64,
  redeem_end_timestamp: i64,
  max_per_user: u64,
  sharing_fee: u64,
  protocol_fee: u64,
  // transferable launchpad
) -> Result<()> {
  let transferable_launchpad = &mut ctx.accounts.transferable_launchpad;

  let current_time = Clock::get().unwrap().unix_timestamp;
  let launchpad_core = transferable_launchpad.get_launchpad_core();
  require!(launchpad_core.total_register == 0, ErrorCode::CannotUpdateLaunchpad);
  require!(launchpad_core.register_start_timestamp > current_time || launchpad_core.register_start_timestamp == 0, ErrorCode::CannotUpdateLaunchpad);

  let launchpad_core: LaunchpadCore = LaunchpadCore {
    is_active,
    whitelist_root,
    max_register,
    max_redeem,
    total_register: 0,
    total_redeem: 0,
    fee_mint,
    max_per_user,
    fee_redeem,
    sharing_fee,
    protocol_fee,
    register_start_timestamp,
    register_end_timestamp,
    redeem_start_timestamp,
    redeem_end_timestamp,
  };

  transferable_launchpad.launchpad_core = launchpad_core;

  emit!(SetTransferableLaunchpadEvent {
    is_active,
    whitelist_root,
    fee_mint,
    fee_redeem,
    register_start_timestamp,
    register_end_timestamp,
    redeem_start_timestamp,
    redeem_end_timestamp,
    max_per_user,
  });

  Ok(())
}
```

5.5. Missing of Launchpad Register Validation

ID	IDX-005
Target	dagora_launchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Low **Impact:** Medium The malicious users can bypass the whitelist checking and create many `user_profile` accounts in order to register the launchpad before the launchpad start time. **Likelihood:** Low It will only occur if the launchpad's owner does not bundle the created and set launchpad data instructions into a single transaction.
Status	Resolved The DAgora team has resolved this issue as suggested by validating that the launchpad has already been activated before users register.

5.5.1. Description

In the launchpad creation process, the launchpad's owner will execute the `create_mintable_launchpad()` or the `create_transferable_launchpad()` functions and the `set_mintable_launchpad()` or the `set_transferable_launchpad()` functions respectively.

lib.rs

```
pub fn create_mintable_launchpad(
  ctx: Context<CreateMintableLaunchpadContext>,
  launchpad_path: Vec<u8>,
  owner: Pubkey,
) -> Result<()> {
  let mintable_launchpad = &mut ctx.accounts.mintable_launchpad;

  mintable_launchpad.nonce = *ctx.bumps.get("mintable_launchpad").unwrap();
  mintable_launchpad.owner = owner;

  emit!(CreateMintableLaunchpadEvent {
    launchpad_path,
    owner
  });

  Ok(())
}
```

lib.rs

```
pub fn set_mintable_launchpad(
  ctx: Context<SetMintableLaunchpadContext>,
  // launchpad core
  is_active: bool,
  whitelist_root: Option<[u8; 32]>,
  max_register: u64,
  max_redeem: u64,
  fee_mint: Pubkey,
  fee_redeem: u64,
  register_start_timestamp: i64,
  register_end_timestamp: i64,
  redeem_start_timestamp: i64,
  redeem_end_timestamp: i64,
  max_per_user: u64,
  sharing_fee: u64,
  protocol_fee: u64,
  // mintable launchpad
  collection_mint: Pubkey,
  name: String,
  symbol: String,
  uri: String,
  seller_fee_basis_points: u16,
  royalty_fee_for_owner: u8,
) -> Result<()> {
  let mintable_launchpad = &mut ctx.accounts.mintable_launchpad;

  let launchpad_core: LaunchpadCore = LaunchpadCore {
    is_active,
    whitelist_root,
    max_register,
    max_redeem,
    total_register: 0,
    total_redeem: 0,
    fee_mint,
    fee_redeem,
    sharing_fee,
    protocol_fee,
    max_per_user,
    register_start_timestamp,
    register_end_timestamp,
    redeem_start_timestamp,
    redeem_end_timestamp,
  };
  mintable_launchpad.launchpad_core = launchpad_core;
  mintable_launchpad.total_nft_redeem = 0;
  mintable_launchpad.collection_mint = collection_mint;
  mintable_launchpad.name = name.clone();
  mintable_launchpad.symbol = symbol.clone();
  mintable_launchpad.uri = uri.clone();
  mintable_launchpad.seller_fee_basis_points = seller_fee_basis_points;
  mintable_launchpad.royalty_fee_for_owner = royalty_fee_for_owner;

  emit!(SetMintableLaunchpadEvent {
    is_active,
    whitelist_root,
    fee_mint,
    fee_redeem,
    register_start_timestamp,
    register_end_timestamp,
    redeem_start_timestamp,
    redeem_end_timestamp,
    max_per_user,
    collection_mint,
    name,
    symbol,
    uri,
    seller_fee_basis_points,
    royalty_fee_for_owner
  });

  Ok(())
}
```

After the launchpad is created, if the launchpad data is already set, the users' addresses should be whitelisted in the `launchpad_core.whitelist_root` before registering; the launchpad and the users can only register after the launchpad starts by using the `register_transferable_launchpad()` or the `register_mintable_launchpad()` functions.

lib.rs

```
pub fn register_mintable_launchpad(
  ctx: Context<RegisterMintableLaunchpadContext>,
  index: u32,
  proofs: Vec<[u8; 32]>,
) -> Result<()> {
  let mintable_launchpad = &mut ctx.accounts.mintable_launchpad;
  let user = &ctx.accounts.user;
  let user_profile = &mut ctx.accounts.user_profile;

  process_register(
    mintable_launchpad,
    user_profile,
    index,
250       user.to_account_info().key(),
251       proofs.clone(),
252     )?;
253
254     emit!(RegisterLaunchpadEvent { index, proofs });
255
256     Ok(())
257   }
```

lib.rs

```
259   pub fn register_transferable_launchpad(
260     ctx: Context<RegisterTransferableLaunchpadContext>,
261     index: u32,
262     proofs: Vec<[u8; 32]>,
263   ) -> Result<()> {
264     let transferable_launchpad = &mut ctx.accounts.transferable_launchpad;
265     let user = &ctx.accounts.user;
266     let user_profile = &mut ctx.accounts.user_profile;
267
268     process_register(
269       transferable_launchpad,
270       user_profile,
271       index,
272       user.to_account_info().key(),
273       proofs.clone(),
274     )?;
275
276     emit!(RegisterLaunchpadEvent { index, proofs });
277
278     Ok(())
279   }
```

The `process_register()` function will validate the users by the `validate_register_time()`, the `validate_max_register()` and the `validate_proof()` functions at lines 642-644.

lib.rs

```
631   fn process_register<
632     'info,
633     Launchpad: LaunchpadProcess + AccountSerialize + AccountDeserialize + Owner +
      Clone,
634   >(
635     launchpad: &mut Account<'info, Launchpad>,
636     user_profile: &mut Account<UserProfile>,
637     index: u32,
638     user_pubkey: Pubkey,
639     proofs: Vec<[u8; 32]>,
) -> Result<()> {
    let mut launchpad_core = launchpad.get_launchpad_core();
    launchpad_core.validate_register_time()?;
    launchpad_core.validate_max_register()?;
    launchpad_core.validate_proof(index, user_pubkey, proofs)?;

    launchpad.set_launchpad_core(launchpad_core);

    user_profile.is_registered = true;
    Ok(())
}
```

On the other hand, if the launchpad information is not set, the validation will be skipped because of the default variable value.

The `validate_register_time()` function will be bypassed since the `register_start_timestamp` and `register_end_timestamp` default values are `0`, by the following lines 62-69.

states.rs

```
60  pub fn validate_register_time(&self) -> Result<()> {
61      let current_time = Clock::get().unwrap().unix_timestamp;
62      require!(
63          self.register_start_timestamp == 0 || current_time >=
    self.register_start_timestamp,
64          ErrorCode::InvalidRegisterTime
65      );
66      require!(
67          self.register_end_timestamp == 0 || current_time <
    self.register_end_timestamp,
68          ErrorCode::InvalidRegisterTime
69      );
70
71      Ok(())
72  }
```

The validation in the `validate_max_register()` function will be skipped because the `max_register` default value is `0`, by the following line 104.

states.rs

```
103  pub fn validate_max_register(&mut self) -> Result<()> {
104      if self.max_register > 0 {
105          require!(self.total_register + 1 <= self.max_register,
     ErrorCode::ReachMaxRegister);
106      }
107
108      self.total_register = self.total_register.checked_add(1).unwrap();
109   Ok(())
110 }
```

Lastly, the `validate_proof()` function will be skipped because the `whitelist_root` value is not set yet, by the following line 89.

states.rs

```
 88 pub fn validate_proof(&self, index: u32, address: Pubkey, proofs: Vec<[u8;
    32]>) -> Result<()> {
 89   if let Some(root) = self.whitelist_root {
 90     let whitelist = WhitelistParams { index, address };
 91     let whitelist_data = whitelist.try_to_vec().unwrap();
 92     let leaf = hash(&whitelist_data[..]);
 93 
 94     require!(
 95       verify_proof(&proofs, &root, &leaf.to_bytes()),
 96       ErrorCode::InvalidProofs
 97     );
 98   }
 99 
100   Ok(())
101 }
```

In conclusion, in the case that the launchpad's owner does not immediately set up the launchpad information after the launchpad has been created, any user can submit the register launchpad transaction to register, which can bypass all validation such as the starting time and whitelisting validation.

5.5.2. Remediation

Inspex suggests bundling the create and set launchpad instructions into a single transaction to prevent users from registering between the create and set launchpad processes.

Alternatively, implement the mechanism to ensure that no user can register the launchpad before the launchpad data is set (`is_active` = true). For example, lines 152 and 175.

contexts.rs

```
145 #[derive(Accounts)]
146 pub struct RegisterMintableLaunchpadContext<'info> {
147   #[account(mut)]
148   pub user: Signer<'info>,
149 
150   #[account(
151       mut,
152       constraint = mintable_launchpad.is_active @ErrorCode::Registered
153   )]
154   pub mintable_launchpad: Account<'info, MintableLaunchpad>,

  #[account(
    mut,
    seeds = [
      user.to_account_info().key().as_ref(),
      mintable_launchpad.to_account_info().key().as_ref()
    ],
    bump = user_profile.nonce,
    constraint = !user_profile.is_registered @ErrorCode::Registered
  )]
  pub user_profile: Account<'info, UserProfile>,
}

#[derive(Accounts)]
pub struct RegisterTransferableLaunchpadContext<'info> {
  #[account(mut)]
  pub user: Signer<'info>,

  #[account(
      mut,
      constraint = transferable_launchpad.is_active @ErrorCode::Registered
  )]
  pub transferable_launchpad: Account<'info, TransferableLaunchpad>,

  #[account(
    mut,
    seeds = [
      user.to_account_info().key().as_ref(),
      transferable_launchpad.to_account_info().key().as_ref()
    ],
    bump = user_profile.nonce,
    constraint = !user_profile.is_registered @ErrorCode::Registered
  )]
  pub user_profile: Account<'info, UserProfile>,
}
```

5.6. Insecure Source of Randomness

ID	IDX-006
Target	dagora_launchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-330: Use of Insufficiently Random Values
Risk	**Severity:** Low **Impact:** Medium A launchpad's owner can control the random result to select a specific NFT. This gives an unfair advantage to the platform users. **Likelihood:** Low There is nothing to restrict the changes from being done; however, this action can only be done by the launchpad's owner and there is no motivation to specify the NFT that was provided by their own. Furthermore, to get the value from manipulating the randomness, the stored NFT must be revealed beforehand, which is an uncommon strategy for the NFT project.
Status	**Acknowledged** The DAgora team has acknowledged this issue; the launchpad’s owner can only do this action, and the launchpad’s owner must be authorized by the DAgora team first.

5.6.1. Description

The `claim_transferable_launchpad()`, the `claim_mintable_launchpad()`, and the `open_mystery_box()` functions are used to claim the NFTs by passing the parameter provided by the launchpad's owner as the random NTF address from the server side.

The `claim_mintable_launchpad()` function will call the `claim_nft()` function and then use `seeds` as the `uri` of NFTs in line 225.

states.rs

```
144 pub fn claim_nft<'info>(
145     &self,
146     launchpad_pubkey: &Pubkey,
147     seeds: Vec<u8>,
148     user: &AccountInfo<'info>,
149     nft_mint_account: &AccountInfo<'info>,
150     launchpad_authority: &AccountInfo<'info>,
151     collection_metadata: &AccountInfo<'info>,
152     collection_mint: &AccountInfo<'info>,
153     collection_master_edition: &AccountInfo<'info>,
    receipt_account: &AccountInfo<'info>,
    token_metadata: &AccountInfo<'info>,
    ata_program: &AccountInfo<'info>,
    token_program: &AccountInfo<'info>,
    system_program: &AccountInfo<'info>,
    sysvar_program: &Sysvar<'info, Rent>,
) -> Result<()> {
    let (nft_mint, nonce): (Pubkey, u8) = Pubkey::find_program_address(&[&seeds], &id());
    let nft_signer_seeds: &[&[u8]] = &[seeds.as_ref(), &[nonce]];

    let (_, authority_nonce): (Pubkey, u8) = get_launchpad_authority(*launchpad_pubkey);

    let launchpad_seeds: &[&[u8]] = &[launchpad_pubkey.as_ref(), &[authority_nonce]];
    let mut creators = Vec::new();

    let owner_launchpad = Creator {
      address: self.owner,
      verified: false,
      share: self.royalty_fee_for_owner,
    };

    creators.push(owner_launchpad);

    let owner_nft = Creator {
      address: user.key(),
      verified: true,
      share: 100u8.checked_sub(self.royalty_fee_for_owner).unwrap(),
    };

    creators.push(owner_nft);

    let collection = Collection {
      key: collection_mint.key(),
      verified: false,
    };

    create_account(
      user,
      nft_mint_account,
      sysvar_program.minimum_balance(82),
      82,
      token_program,
      &[nft_signer_seeds],
    )?;
    create_token_mint(
      nft_mint_account,
      &sysvar_program.to_account_info(),
      0,
      launchpad_authority.to_account_info().key(),
      COption::None,
      &[],
    )?;
    create_ata(
      receipt_account,
      user,
      nft_mint_account,
      user,
      ata_program,
      token_program,
      system_program,
      &sysvar_program.to_account_info(),
      &[],
    )?;
    mint_token(
      launchpad_authority,
      nft_mint_account,
      receipt_account,
      1,
      &[launchpad_seeds],
    )?;
    let mut uri = self.uri.clone().to_owned();
    uri.push_str(&nft_mint.to_string());

    let mut name = self.name.clone().to_owned();
    name.push_str(&" #".to_string());
    name.push_str(&self.total_nft_redeem.to_string());
    create_token_metadata(
      token_metadata,
      nft_mint_account,
      launchpad_authority,
      user,
      user,
      name,
      self.symbol.clone(),
      uri,
      Some(creators),
      self.seller_fee_basis_points,
      true,
      false,
      Some(collection),
      None,
      system_program,
      &sysvar_program.to_account_info(),
      &[launchpad_seeds],
    )?;
    verify_collection_for_token(
      token_metadata,
      launchpad_authority,
      user,
      collection_mint,
      collection_metadata,
      collection_master_edition,
      None,
      &[launchpad_seeds],
    )?;

    Ok(())
}
```

This results in some users can specifically choose the NFTs using the **seeds** which generate from the server side.

Moreover, in the `claim_transferable_launchpad()` and the `open_mystery_box()` functions as shown below:

lib.rs

```
pub fn claim_transferable_launchpad(
    ctx: Context<ClaimTransferableLaunchpadContext>,
    nft_mint: Pubkey,
    owner_signature: [u8; 64]
) -> Result<()> {
    let transferable_launchpad = &ctx.accounts.transferable_launchpad;
    let launchpad_authority = &ctx.accounts.launchpad_authority;
    let user_profile = &mut ctx.accounts.user_profile;
    let launchpad_nft_token_account =
&ctx.accounts.launchpad_nft_token_account;
    let user_nft_token_account = &ctx.accounts.user_nft_token_account;

    let ix: Instruction = load_instruction_at_checked(0,
&ctx.accounts.sysvar_program)?;

    require!(ix.program_id == ED25519_ID,
ErrorCode::InvalidValidateSignInstruction);
    require!(ix.accounts.len() == 0,
ErrorCode::InvalidValidateSignInstruction);

    let message = MessageRandom {
      root: user_profile.key(),
      nft_mint
    }.try_to_vec().unwrap();

    let hashed_message = hash(&message).to_bytes();
    require!(ix.data.len() == (16 + 64 + 32 + hashed_message.len()),
ErrorCode::InvalidValidateSignInstruction);
    check_ed25519_data(&ix.data, transferable_launchpad.owner.as_ref(),
&hashed_message, &owner_signature)?;

    let launchpad_pubkey = &transferable_launchpad.to_account_info().key();

    user_profile.before_claim()?;

    transferable_launchpad.claim_nft(
      launchpad_pubkey,
      launchpad_authority,
      launchpad_nft_token_account,
      user_nft_token_account,
    )?;

    emit!(ClaimLaunchpadEvent { nft_mint });

    Ok(())
}

pub fn open_mystery_box(
    ctx: Context<OpenMysteryBoxContext>,
    mystery_nft_mint: Pubkey,
    owner_signature: [u8; 64]
) -> Result<()> {
    let user = &ctx.accounts.user;
    let mintable_launchpad = &mut ctx.accounts.mintable_launchpad;
    let launchpad_authority = &ctx.accounts.launchpad_authority;
    let box_mint = &ctx.accounts.box_mint;
    let box_metadata = &ctx.accounts.box_metadata;
    let user_box_token_account = &ctx.accounts.user_box_token_account;
    let launchpad_box_token_account =
&ctx.accounts.launchpad_box_token_account;
    let launchpad_mystery_token_account =
&ctx.accounts.launchpad_mystery_token_account;
    let user_mystery_token_account = &ctx.accounts.user_mystery_token_account;

    let ix: Instruction = load_instruction_at_checked(0,
&ctx.accounts.sysvar_program)?;

    require!(ix.program_id == ED25519_ID, ErrorCode::SigVerificationFailed);
    require!(ix.accounts.len() == 0, ErrorCode::SigVerificationFailed);

    let message = MessageRandom {
      root: box_mint.key(),
      nft_mint: mystery_nft_mint
    }.try_to_vec().unwrap();

    let hashed_message = hash(&message[..]).to_bytes();
    // validate messgae len
    require!(ix.data.len() == (16 + 64 + 32 + hashed_message.len()), ErrorCode::SigVerificationFailed);
    check_ed25519_data(&ix.data, mintable_launchpad.owner.as_ref(), &hashed_message, &owner_signature)?;

    let box_metadata: Metadata = Metadata::from_account_info(&box_metadata).unwrap();

    if let Some(collection) = box_metadata.collection {
      require!(
        collection.key == mintable_launchpad.collection_mint,
        ErrorCode::CollectionMismatch
      );
    } else {
      return Err(ErrorCode::CollectionMismatch.into());
    }

    let launchpad_pubkey = &mintable_launchpad.to_account_info().key();
    let (_, authority_nonce): (Pubkey, u8) = get_launchpad_authority(launchpad_pubkey.clone());
    let seeds: &[&[_]] = &[launchpad_pubkey.as_ref(), &[authority_nonce]];

    transfer_token(
      user,
      user_box_token_account,
      launchpad_box_token_account,
      1,
      &[],
    )?;
    burn_token(
      launchpad_authority,
      box_mint,
      launchpad_box_token_account,
      1,
      &[seeds],
    )?;

    transfer_token(
496         launchpad_authority,
497         launchpad_mystery_token_account,
498         user_mystery_token_account,
499         1,
500         &[seeds],
501     )?;
502 
503     emit!(OpenMysteryBoxEvent {});
504 
505     Ok(())
506 }
```

The `nft_mint` and the `mystery_nft_mint` parameters are validated in lines 417 and 463, which are signed by the owner. The signed message contains both a random NFT index and the user's account. This prevents the signed message from being used by the other users to claim the specific NFT.

However, the random result can be a specific NFT for some users. This is because the `nft_mint` and the `mystery_nft_mint` can be picked and signed by the owner. This results in an unfair advantage for the platform's users.

5.6.2. Remediation

Inspex suggests applying the provably fair and verifiable randomness of the NFT random instead of the centralization random from the off-chain, which the platform can control.

For example, implementing the VRF as the randomness source for the NFT random.

5.7. Improper Fee Enforcement

ID	IDX-007
Target	dagora_launchpad
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The launchpad’s owner will either lose all profits or have to pay an additional fee to the platform. **Likelihood:** Low It is extremely unlikely that the launchpad's owner or the platform will impose an unreasonable fee that will harm both the launchpad's profitability and the platform's reputation.
Status	Resolved The DAgora team has resolved this issue by ensuring that the `total_system_fee` is less than the `total_redeem_fee` and adding the boundary of `protocol_fee` up to 20%.

5.7.1. Description

During the redemption phase, the registered user could redeem their via the `redeem_mintable_launchpad()` function or the `redeem_transferable_launchpad()` function, which both will call the `process_redeem()` function as shown in lines 296 and 330:

lib.rs

```
281 pub fn redeem_mintable_launchpad<'a>(
282   ctx: Context<'_, '_, '_, 'a, RedeemMintableLaunchpadContext<'a>>,
283   amount: u64,
284 ) -> Result<()> {
285   let user = &ctx.accounts.user;
286   let mintable_launchpad = &mut ctx.accounts.mintable_launchpad;
287   let launchpad_authority = &ctx.accounts.launchpad_authority;
288   let user_profile = &mut ctx.accounts.user_profile;
289 
290   let user_fee_token_account = &ctx.accounts.user_fee_token_account;
291   let launchpad_fee_token_account = &ctx.accounts.launchpad_fee_token_account;
292   let protocol_fee_token_account = &ctx.accounts.protocol_fee_token_account;
293 
294   let launchpad_key = &mintable_launchpad.to_account_info().key();
295 
```

```
  process_redeem(
    mintable_launchpad,
    &launchpad_key,
    user_profile,
    &user.to_account_info(),
    launchpad_authority,
    user_fee_token_account,
    launchpad_fee_token_account,
    protocol_fee_token_account,
    amount,
  )?;

  emit!(RedeemLaunchpadEvent { amount });

  Ok(())
}
```

lib.rs

```
pub fn redeem_transferable_launchpad(
  ctx: Context<RedeemTransferableLaunchpadContext>,
  amount: u64,
) -> Result<()> {
  let user = &ctx.accounts.user;
  let transferable_launchpad = &mut ctx.accounts.transferable_launchpad;
  let launchpad_authority = &ctx.accounts.launchpad_authority;
  let user_profile = &mut ctx.accounts.user_profile;

  let user_fee_token_account = &ctx.accounts.user_fee_token_account;
  let launchpad_fee_token_account = &ctx.accounts.launchpad_fee_token_account;
  let protocol_fee_token_account = &ctx.accounts.protocol_fee_token_account;

  let launchpad_key = &transferable_launchpad.to_account_info().key();

  // we cover limit by total nft in launchpad

  process_redeem(
    transferable_launchpad,
    &launchpad_key,
    user_profile,
    &user.to_account_info(),
    launchpad_authority,
    user_fee_token_account,
    launchpad_fee_token_account,
    protocol_fee_token_account,
    amount,
  )?;

342    emit!(RedeemLaunchpadEvent { amount });
343
344    Ok(())
345  }
```

The `process_redeem()` function is used to process the fee that the registered user must pay to redeem the NFT.

The `total_fee` state is the total fee that the user must pay, which is calculated by multiplying the amount by the `launchpad_core.fee_redeem` state as shown in line 595.

lib.rs

```
569  fn process_redeem<
570    'info,
571    Launchpad: LaunchpadProcess + AccountSerialize + AccountDeserialize + Owner +
     Clone,
572  >(
573    launchpad: &mut Account<Launchpad>,
574    launchpad_key: &Pubkey,
575    user_profile: &mut Account<UserProfile>,
576    user: &AccountInfo<'info>,
577    launchpad_authority: &AccountInfo<'info>,
578    user_fee_token_account: &AccountInfo<'info>,
579    launchpad_fee_token_account: &AccountInfo<'info>,
580    protocol_fee_token_account: &AccountInfo<'info>,
581    amount: u64,
582  ) -> Result<()> {
583    let mut launchpad_core: LaunchpadCore = launchpad.get_launchpad_core();
584    launchpad_core.validate_redeem_time()?;
585    launchpad_core.validate_max_redeem(amount)?;
586
587    if launchpad_core.max_per_user > 0 {
588      require!(
589        user_profile.total_nft_redeem.checked_add(amount).unwrap() <=
     launchpad_core.max_per_user,
590        ErrorCode::ReachMaximumPerUser
591      );
592    }
593    launchpad.set_launchpad_core(launchpad_core);
594
595    let total_fee = amount.checked_mul(launchpad_core.fee_redeem).unwrap();
596
```

When the `total_fee` state is greater than zero, the `procotol_fee` is calculated by multiplying the `total_fee` state with the `launchpad_core.protocol_fee` state as shown below in line 598.

Moreover, the `protocol_fee` is added by the `launchpad_core.sharing_fee` state and transferred to the`launchpad_fee_token_account` account as shown below in line 603.

lib.rs

```
597 if total_fee > 0 {
598   let mut protocol_fee = total_fee
599     .checked_mul(launchpad_core.protocol_fee)
600     .unwrap()
601     .checked_div(10000)
602     .unwrap();
603   protocol_fee = protocol_fee.checked_add(launchpad_core.sharing_fee).unwrap();
604   transfer_token(
605     user,
606     &user_fee_token_account,
607     &launchpad_fee_token_account,
608     total_fee,
609     &[],
610   )?;
611
612   if protocol_fee > 0 {
613     let (_, authority_nonce): (Pubkey, u8) =
    get_launchpad_authority(*launchpad_key);
614     let seeds: &[&[_]] = &[launchpad_key.as_ref(), &[authority_nonce]];
615
616     transfer_token(
617       launchpad_authority,
618       launchpad_fee_token_account,
619       protocol_fee_token_account,
620       protocol_fee,
621       &[seeds],
622     )?;
623   }
624 }
```

If the `protocol_fee` is greater than zero, it will transfer the fee from the `launchpad_fee_token_account` account to the `protocol_fee_token_account` account.

In some cases, the `protocol_fee` that transfers from the `launchpad_fee_token_account` account to the `protocol_fee_token_account` account could be greater than the `total_fee` that the registered user transfers to the `launchpad_fee_token_account`, for example:

The registered user transfers to the `total_fee` only 300.

Variable	Value
launchpad_core.fee_redeem	100
amount	3

total_fee	3 * 100 = 300

The program transfers 450 (`protocol_fee`) from the `launchpad_fee_token_account` account to the `protocol_fee_token_account` account, which is greater than the registered user transfer to the `launchpad_fee_token_account` account which is just 300(`total_fee`).

Variable	Value
protocol_fee	50%
sharing_fee	300
protocol_fee	(50% of 300) + 300 = 450

This results in the exceeded fee being drained from the `launchpad_fee_token_account` account instead.

5.7.2. Remediation

Inspex suggests insisting the user pay the extra fee as the most valuable of the `total_fee` and the `protocol_fee` to prevent the launchpad from losing its profits, For example in lines 607-609:

lib.rs

```
595 let total_fee = amount.checked_mul(launchpad_core.fee_redeem).unwrap();
596
597 if total_fee > 0 {
598   let mut protocol_fee = total_fee
599     .checked_mul(launchpad_core.protocol_fee)
600     .unwrap()
601     .checked_div(10000)
602     .unwrap();
603   protocol_fee = protocol_fee.checked_add(launchpad_core.sharing_fee).unwrap();
604
605   let mut fee_to_pay = total_fee;
606
607   if (protocol_fee > total_fee){
608     fee_to_pay = protocol_fee;
609   }
610
611   transfer_token(
612     user,
613     &user_fee_token_account,
614     &launchpad_fee_token_account,
615     fee_to_pay,
616     &[],
617   )?;
618
619   if protocol_fee > 0 {
620     let (_, authority_nonce): (Pubkey, u8) =
get_launchpad_authority(*launchpad_key);
  let seeds: &[&[_]] = &[launchpad_key.as_ref(), &[authority_nonce]];

  transfer_token(
   launchpad_authority,
   launchpad_fee_token_account,
   protocol_fee_token_account,
   fee_to_pay,
   &[seeds],
  )?;
 }
}
```

5.8. Smart Contract with Unpublished Source Code

ID	IDX-008
Target	dagora_solana
Category	General Smart Contract Vulnerability
CWE	CWE-1006: Bad Coding Practices
Risk	**Severity:** Low **Impact:** Medium The logic of the smart contract may not align with the user's understanding, causing undesired actions to be taken when the user interacts with the smart contract. **Likelihood:** Low The possibility for the users to misunderstand the functionalities of the contract is not very high with the help of the documentation and user interface.
Status	Acknowledged The Coin98 team has acknowledged this issue and decided not to publish the source code because the team wants to protect their intellectual property.

5.8.1. Description

The smart contract source code is not publicly published, so the users will not be able to easily verify the correctness of the functionalities and the logic of the smart contract by themselves. Therefore, it is possible that the user's understanding of the smart contract does not align with the actual implementation, leading to undesired actions on interacting with the smart contract.

5.8.2. Remediation

Inspex suggests publishing the contract source code through a public code repository or verifying the smart contract source code on the blockchain explorer so that the users can easily read and understand the logic of the smart contract by themselves.

6. Appendix

6.1. About Inspex

CYBERSECURITY
PROFESSIONAL
SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspex

CYBERSECURITY PROFESSIONAL SERVICE