Solana Marketplace

Smart Contract Audit Report
Prepared for DAgora

Date Issued:	Aug 21, 2023
Project ID:	AUDIT2022039
Version:	v2.0
Confidentiality Level:	Public

inspex
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022039
Version	v2.0
Client	DAgora
Project	Solana Marketplace
Auditor(s)	Peeraphut Punsuwan Puttimet Thammasaeng Ronnachai Chaipha Sorawish Laovakul
Author(s)	Wachirawit Kanpanluk
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
2.0	Aug 21,2023	Update issue	Wachirawit Kanpanluk
1.0	Jun 23, 2022	Full report	Sorawish Laovakul

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 Solana Marketplace smart contracts between Jun 9, 2022 and Jun 16, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Solana Marketplace 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 4 high, 1 medium, 2 low-severity issues. With the project team's prompt response, 4 high and 1 low-severity issues were resolved or mitigated in the reassessment, while 1 medium and 1 low-severity issues were acknowledged by the team. Therefore, Inspex trusts that Solana Marketplace 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.

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 Solana Marketplace is the NFT Marketplace in Solana. It allows anyone to buy, sell, and auction NFTs on the Solana.

The platform allows the NFT to be freely bought and sold by all the platform users in a single NFT or multiple NFTs in one transaction. The platform also provides the royalty fee for the NFT creators, helping them to gain their revenue.

Scope Information:

Project Name	Solana Marketplace
Website	https://dagora.xyz/
Smart Contract Type	Solana Program
Chain	Solana
Programming Language	Rust
Category	Marketplace

Audit Information:

Audit Method	Whitebox
Audit Date	Jun 9, 2022 - Jun 16, 2022
Reassessment Date	Jun 21, 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

Program	Bytecode SHA256 Hash
dagora_solana	6195f01c687fa47b23f08d26b1c4e6643348c6171599b9f4ac3e82a7a6cb0a73

Reassessment

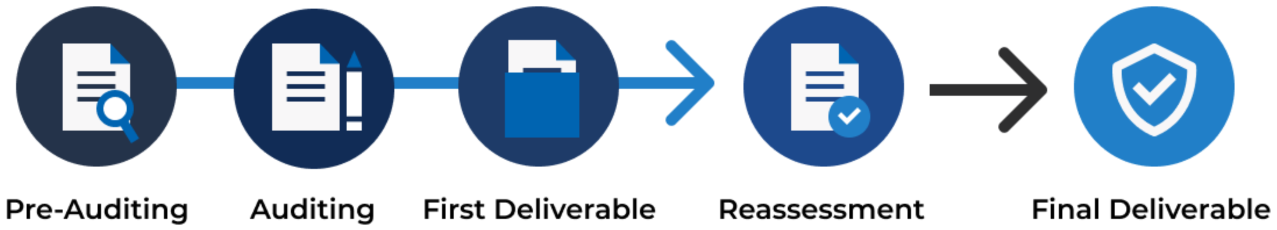
Program	Bytecode SHA256 Hash
dagora_solana	d48a95ca103c3a208e587f8dc21f95d87ed7e83c4500e6ffd25db0b87a218566

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**.

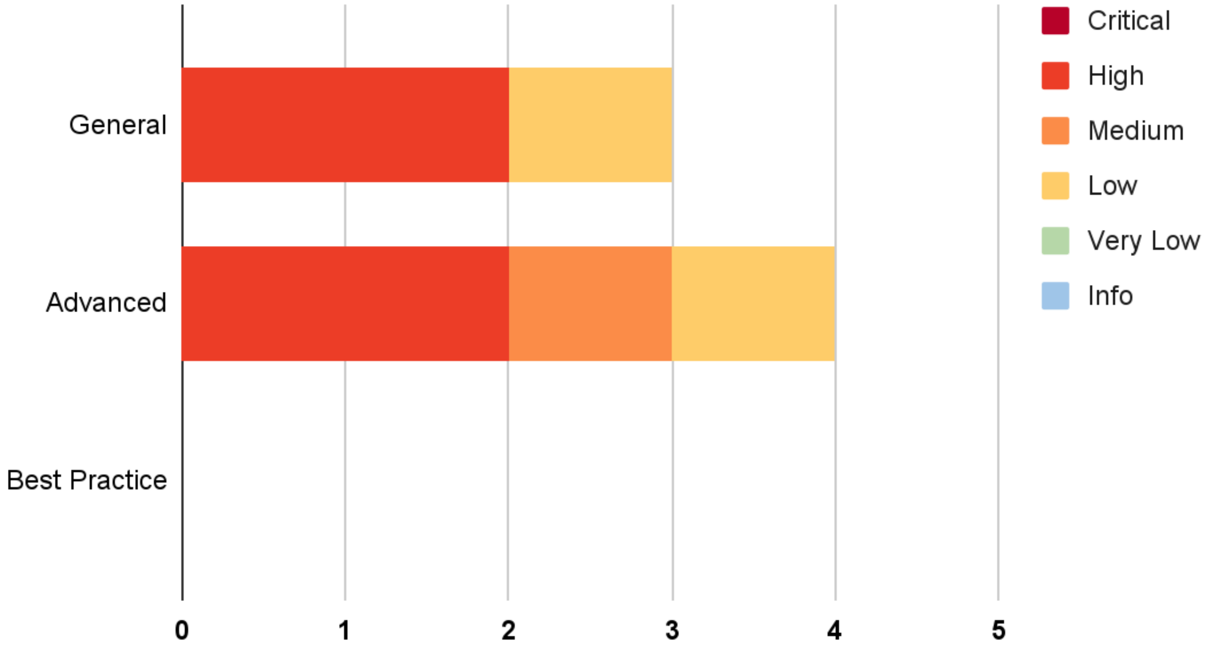
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

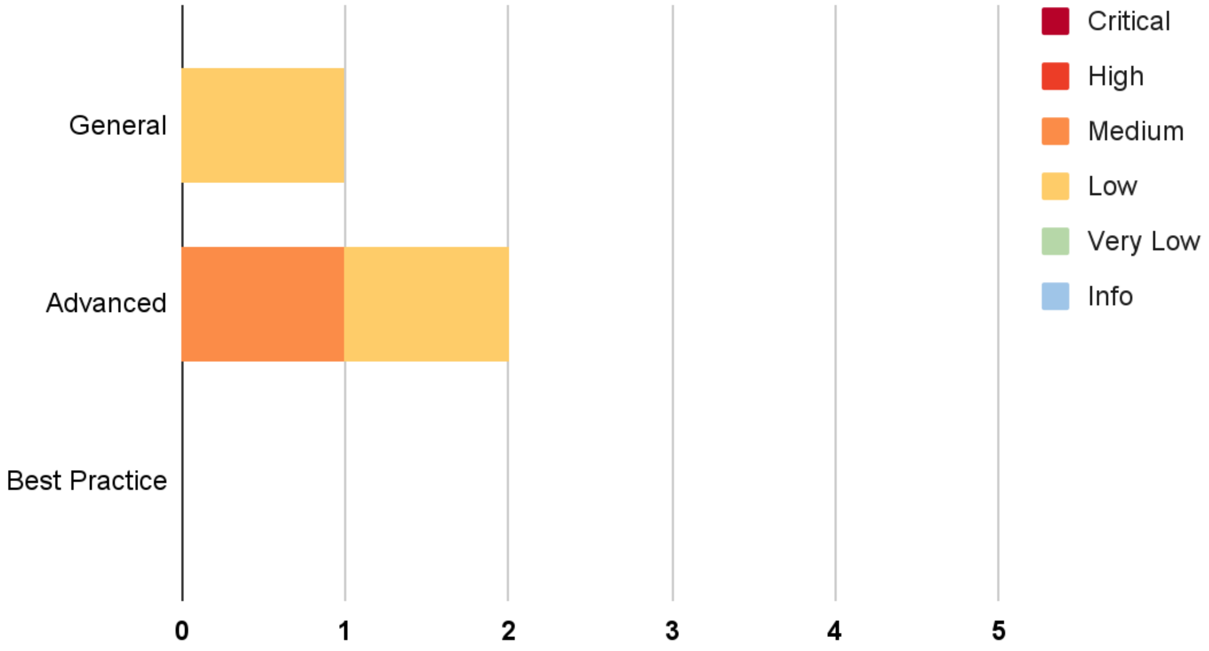
4. Summary of Findings

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	Improper Offer Acceptance	Advanced	High	Resolved
IDX-002	Lack of NFT Listing State Validation	Advanced	High	Resolved
IDX-003	Upgradability of Solana Program	General	High	Resolved *
IDX-004	Centralized Control of State Variable	General	High	Resolved *
IDX-005	Design Flaw in Auction Mechanism	Advanced	Medium	Acknowledged
IDX-006	Unbound Configuration Parameter	Advanced	Low	Acknowledged
IDX-007	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. Improper Offer Acceptance

ID	IDX-001
Target	dagora_solana
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: High** **Impact: High** The attacker, as the NFT seller, can sell their NFT to any buyer by accepting an offer from them. However, the NFT can be bought back from the attacker instantly at a cheap price, resulting in the attacker gaining free money. **Likelihood: Medium** This attack scenario requires a buyer who offers a price that is cheaper than the selling price. After there is an offer, it is likely that the attacker will execute this scenario since there is a very low cost of executing it.
Status	Resolved The DAgora team has resolved this issue by implementing the new vault status design to validate the vault state. This issue has been resolved.

5.1.1. Description

The `create_offer()` function on the DAgora NFT marketplace allows a buyer to make an offer to purchase the NFT at a cheaper price than the selling price.

lib.rs

```
pub fn create_offer(
    ctx: Context<CreateOfferContext>,
    _package_token: Pubkey,
    amount: u64,
) -> Result<()> {
    msg!("DAgora Marketplace: Create Offer Instruction");
    let buyer = &ctx.accounts.buyer;
    let order_account = &mut ctx.accounts.order_account;

    let vault_authority_account = &ctx.accounts.vault_authority_account;
    let listing_account = &ctx.accounts.listing_account;
    let buyer_package_token_account =
&ctx.accounts.buyer_package_token_account;

    order_account.order_type = OrderType::Offer;
    order_account.nonce = *ctx.bumps.get("order_account").unwrap();
    order_account.vault_account = listing_account.vault_account;
    order_account.buyer = *buyer.to_account_info().key;
    order_account.listing_account = *listing_account.to_account_info().key;
    order_account.amount = amount;

    if amount >= listing_account.amount {
        order_account.status = OrderStatus::Accept;
    } else {
        order_account.status = OrderStatus::Created;
    }

    approve_token(&buyer.to_account_info(),
&buyer_package_token_account.to_account_info(),
&vault_authority_account.to_account_info(), amount, &[])?;

    emit!(CreateOfferEvent{
        amount
    });

    Ok(())
}
```

In the case that the attacker is a seller, when there is an offer coming to them, the attacker can offer themselves with a lower price through the `create_offer()` function, then accept both offers with the `accept_offer()` function.

lib.rs

```
pub fn accept_offer(
    ctx: Context<AcceptOfferContext>
) -> Result<()> {
    msg!("DAgora Marketplace: Accept Offer Instruction");
    let order_account = &mut ctx.accounts.order_account;

    order_account.status = OrderStatus::Accept;

    emit!(AcceptOfferEvent{});
    Ok(())
}
```

The `execute_order()` function allows any user to execute this function to transfer the NFT or vault account ownership to the buyer and receive the buyer's token as an exchange.

To attack, the attacker will offer themselves with the lower price bid and then the attacker will execute the highest value offer. After executing the highest bid, the vault ownership will be transferred to the buyer and the buyer's token will be transferred to the attacker. The attacker will then execute the lowest bid, the vault ownership will be transferred back to the attacker, and the token will be transferred back to the buyer for the first time.

lib.rs

```rust
pub fn execute_order<'info>(
    ctx: Context<'_, '_, '_, 'info, ExecuteOrderContext<'info>>,
    package_token: Pubkey,
) -> Result<()>{
    msg!("DAgora Marketplace: Execute Order Instruction");

    let package_account = &ctx.accounts.package_account;
    let order_account = &ctx.accounts.order_account;

    let vault_account = &ctx.accounts.vault_account;
    let vault_account_key = vault_account.key();

    let vault_authority_account = &ctx.accounts.vault_authority_account;

    let buyer_package_token_account = &ctx.accounts.buyer_package_token_account;
    let fee_owner_token_address = &ctx.accounts.fee_owner_token_address;
    let seller_package_token_account = &ctx.accounts.seller_package_token_account;

    let seeds: &[&[_]] = &[AUTHORITY_SEED, &vault_account_key.as_ref(), &[vault_account.authority_nonce]];

    let (system_fee, amount_after_sub_system_fee) = order_account.split_amount(package_account.market_fee, package_account.claim_fee);

    // transfer system fee
    transfer_token(
        vault_authority_account,
        &buyer_package_token_account.to_account_info(),
        &fee_owner_token_address.to_account_info(),
        system_fee,
        &[seeds]
    )?;

    let account_iter = &mut ctx.remaining_accounts.iter();

    let mut total_royalty_fee_transferred: u64 = 0;

    let vault_account = &ctx.accounts.vault_account;

    if vault_account.vault_type == VaultType::SingleItem {
        let from_nft_account_info = next_account_info(account_iter)?;
        let to_nft_account_info = next_account_info(account_iter)?;
        let metadata_account_info = next_account_info(account_iter)?;

        require!(from_nft_account_info.key() == get_associated_token_address(&vault_account.owner, &vault_account.nft_mints[0]), ErrorCode::InvalidSellerNftTokenAccount);
        require!(to_nft_account_info.key() == get_associated_token_address(&order_account.buyer, &vault_account.nft_mints[0]), ErrorCode::InvalidBuyerNftTokenAccount);
        require!(metadata_account_info.key() == find_metadata_account(&vault_account.nft_mints[0]).0, ErrorCode::InvalidMetadataAccount);

        transfer_token(vault_authority_account, &from_nft_account_info, &to_nft_account_info, 1, &[seeds])?;

        if !metadata_account_info.data_is_empty() {
            total_royalty_fee_transferred = transfer_royalty_fee(account_iter, metadata_account_info, vault_authority_account, buyer_package_token_account, &package_token, amount_after_sub_system_fee, &[seeds])?;
        }
    } else {
        if vault_account.total_royalty_fee > 0 {
            let vault_royalty_fee_owner = next_account_info(account_iter)?;
            require!(vault_royalty_fee_owner.key() == get_associated_token_address(&vault_authority_account.key(), &package_token), ErrorCode::InvalidVaultRoyaltyFeeOwner);
            let royalty_fee = package_account.royalty_fee;

            total_royalty_fee_transferred = amount_after_sub_system_fee.checked_mul(royalty_fee.into()).unwrap().checked_div(PERCENT.into()).unwrap();

            transfer_token(vault_authority_account, buyer_package_token_account, vault_royalty_fee_owner, total_royalty_fee_transferred, &[seeds])?;
        }
    }

    // transfer amount to seller
    transfer_token(
        vault_authority_account,
        &buyer_package_token_account.to_account_info(),
        &seller_package_token_account.to_account_info(),

amount_after_sub_system_fee.checked_sub(total_royalty_fee_transferred).unwrap()
,
        &[seeds]
    )?;

    let vault_account = &mut ctx.accounts.vault_account;

    vault_account.owner = order_account.buyer;
    vault_account.status = VaultStatus::Sold;

    Ok(())
}
```

As a result, the attacker can gain the profit without losing the NFT. This can be summarized as the example step below.

1. The attacker sells NFT as multi-typing for $150.
2. Alice offered to buy the NFT for $100.
3. The attacker offered to buy the NFT for $1.
4. The attacker accepts Alice's order and the attacker's order.
5. The attacker executes Alice's order to take $100 and transfer the vault account ownership to Alice.
6. The attacker executes the attacker's order to transfer $1 to Alice and transfers the vault account ownership back to the attacker.

In this scenario, the attacker keeps the NFT but makes a profit of $99.

5.1.2. Remediation

Inspex suggests adding the condition to verify that the `vault_account.status` is `OnSale` at the `execute_order()` function, so whenever the buying process has been executed, the `vault_account.status` state will be changed to `VaultStatus::Sold`. Therefore, the `execute_order()` function will not be able to be executed again.

instructions.rs

```
#[derive(Accounts)]
#[instruction(package_token: Pubkey)]
pub struct ExecuteOrderContext<'info> {
  #[account(
    seeds = [
      package_token.key().as_ref()
    ],
    bump = package_account.nonce,
    constraint = package_account.is_active @ErrorCode::PackageNotActiveYet,
  )]
  pub package_account: Box<Account<'info, PackageInfo>>,

  #[account(
    mut,
    constraint = vault_account.package_token == package_token @ErrorCode::InvalidPackageAccount,
    constraint = vault_account.status == VaultStatus::OnSale @ErrorCode::InvalidVaultStatus,
  )]
  pub vault_account: Account<'info, VaultInfo>,

  /// CHECK: Authority of Vault account
  #[account(
    seeds = [
      AUTHORITY_SEED,
      vault_account.to_account_info().key.as_ref()
    ],
    bump = vault_account.authority_nonce
  )]
  pub vault_authority_account: AccountInfo<'info>,

  #[account(
    constraint = order_account.vault_account == *vault_account.to_account_info().key @ErrorCode::InvalidVaultAccount,

    constraint = order_account.status == OrderStatus::Accept @ErrorCode::InvalidOrderStatus,
  )]
  pub order_account: Box<Account<'info, OrderInfo>>,

  /// CHECK: This is not dangerous because we don't read or write from this account
  #[account(
    mut,
    address = get_associated_token_address(&order_account.buyer, &package_token)
  )]
  pub buyer_package_token_account: AccountInfo<'info>,

  /// CHECK: This is not dangerous because we don't read or write from this account
  #[account(
    mut,
    address = get_associated_token_address(&package_account.fee_owner, &package_token)
  )]
  pub fee_owner_token_address: AccountInfo<'info>,

  /// CHECK: This is not dangerous because we don't read or write from this account
  #[account(
    mut,
    address = get_associated_token_address(&vault_account.owner, &package_token)
  )]
  pub seller_package_token_account: AccountInfo<'info>,

  /// CHECK: We have checked address
  #[account(
    address = TOKEN_PROGRAM_ID
  )]
  pub token_program: AccountInfo<'info>,
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.2. Lack of NFT Listing State Validation

ID	IDX-002
Target	dagora_solana
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The seller can withdraw the NFT before calling the `execute_order()` function, resulting in transferring the empty vault (no NFT) to the buyer. **Likelihood:** Medium This issue requires an offer to the attacker's listed NFT from the buyers. All of the funds in the buyer's offer will be stolen.
Status	Resolved The DAgora team has resolved this issue by implementing the new vault status design to validate the vault state. This issue has been resolved.

5.2.1. Description

The NFT owner can sell the NFT by creating a vault account with the `create_multi_items_vault()` function and transferring the NFT token to the vault account with the `deposit_item()` function.

lib.rs

```
pub fn create_multi_items_vault(
    ctx: Context<CreateMultiItemsVaultContext>,
    _vault_path: Vec<u8>,
    authority_nonce: u8,
    _mint_size: u16,
) -> Result<()> {
    msg!("DAgora Marketplace: Create Multi Items Vault Instruction");

    let owner = &ctx.accounts.owner;
    let vault = &mut ctx.accounts.vault;

    vault.authority_nonce = authority_nonce;
    vault.owner = owner.key();
    vault.vault_type = VaultType::MultiItems;

    vault.nft_mints = Vec::new();
    vault.status = VaultStatus::Create;
    vault.total_royalty_fee = 0;

    Ok(())
}

pub fn deposit_item(
    ctx: Context<DepositItemContext>,
    nft_mint: Pubkey,
) -> Result<()> {
    msg!("DAgora Marketplace: Deposit Item To Vault Instruction");
    let owner = &ctx.accounts.owner;
    let vault_account = &mut ctx.accounts.vault_account;
    let owner_token_account = &ctx.accounts.owner_token_account;
    let vault_token_account = &ctx.accounts.vault_token_account;
    let metadata_account = &ctx.accounts.metadata_account;

    transfer_token(owner, owner_token_account, vault_token_account, 1, &[])?;

    let nft_mints = &mut vault_account.nft_mints;
    nft_mints.push(nft_mint);

    if !metadata_account.data_is_empty() {
        let metadata: Metadata = Metadata::from_account_info(metadata_account).unwrap();

        if metadata.data.seller_fee_basis_points > 0 {
            vault_account.total_royalty_fee = vault_account.total_royalty_fee.checked_add(metadata.data.seller_fee_basis_points.into()).unwrap();
        }
    }

    emit!(DepositItemEvent{
        nft_mint
    });

    Ok(())
}
```

When the buyer uses the `create_offer()` function to make an offer to the NFT that is on selling, the buyer's token account will be approved to the vault account in order to transfer the token later when the selling process is completed.

lib.rs

```
pub fn create_offer(
    ctx: Context<CreateOfferContext>,
    _package_token: Pubkey,
    amount: u64,
) -> Result<()> {
    msg!("DAgora Marketplace: Create Offer Instruction");
    let buyer = &ctx.accounts.buyer;
    let order_account = &mut ctx.accounts.order_account;

    let vault_authority_account = &ctx.accounts.vault_authority_account;
    let listing_account = &ctx.accounts.listing_account;
    let buyer_package_token_account =
&ctx.accounts.buyer_package_token_account;

    order_account.order_type = OrderType::Offer;
    order_account.nonce = *ctx.bumps.get("order_account").unwrap();
    order_account.vault_account = listing_account.vault_account;
    order_account.buyer = *buyer.to_account_info().key;
    order_account.listing_account = *listing_account.to_account_info().key;
    order_account.amount = amount;

    if amount >= listing_account.amount {
        order_account.status = OrderStatus::Accept;
    } else {
        order_account.status = OrderStatus::Created;
    }

    approve_token(&buyer.to_account_info(),
&buyer_package_token_account.to_account_info(),
&vault_authority_account.to_account_info(), amount, &[])?;

    emit!(CreateOfferEvent{
        amount
    });

    Ok(())
}
```

To complete the selling process, the `accept_offer()` and the `execute_order()` functions must be executed respectively in order to transfer the token to the seller and change the vault ownership to the buyer.

lib.rs

```rust
pub fn accept_offer(
    ctx: Context<AcceptOfferContext>
) -> Result<()> {
    msg!("DAgora Marketplace: Accept Offer Instruction");
    let order_account = &mut ctx.accounts.order_account;

    order_account.status = OrderStatus::Accept;

    emit!(AcceptOfferEvent{});
    Ok(())
}

pub fn execute_order<'info>(
    ctx: Context<'_, '_, '_, 'info, ExecuteOrderContext<'info>>,
    package_token: Pubkey,
) -> Result<()>{
    msg!("DAgora Marketplace: Execute Order Instruction");

    let package_account = &ctx.accounts.package_account;
    let order_account = &ctx.accounts.order_account;

    let vault_account = &ctx.accounts.vault_account;
    let vault_account_key = vault_account.key();

    let vault_authority_account = &ctx.accounts.vault_authority_account;

    let buyer_package_token_account = &ctx.accounts.buyer_package_token_account;
    let fee_owner_token_address = &ctx.accounts.fee_owner_token_address;
    let seller_package_token_account = &ctx.accounts.seller_package_token_account;

    let seeds: &[&[_]] = &[AUTHORITY_SEED, &vault_account_key.as_ref(), &[vault_account.authority_nonce]];

    let (system_fee, amount_after_sub_system_fee) = order_account.split_amount(package_account.market_fee, package_account.claim_fee);

    // transfer system fee
    transfer_token(
        vault_authority_account,
        &buyer_package_token_account.to_account_info(),
        &fee_owner_token_address.to_account_info(),
        system_fee,
        &[seeds]
    )?;

    let account_iter = &mut ctx.remaining_accounts.iter();

    let mut total_royalty_fee_transferred: u64 = 0;

    let vault_account = &ctx.accounts.vault_account;

    if vault_account.vault_type == VaultType::SingleItem {
        let from_nft_account_info = next_account_info(account_iter)?;
        let to_nft_account_info = next_account_info(account_iter)?;
        let metadata_account_info = next_account_info(account_iter)?;

        require!(from_nft_account_info.key() == get_associated_token_address(&vault_account.owner, &vault_account.nft_mints[0]), ErrorCode::InvalidSellerNftTokenAccount);
        require!(to_nft_account_info.key() == get_associated_token_address(&order_account.buyer, &vault_account.nft_mints[0]), ErrorCode::InvalidBuyerNftTokenAccount);
        require!(metadata_account_info.key() == find_metadata_account(&vault_account.nft_mints[0]).0, ErrorCode::InvalidMetadataAccount);

        transfer_token(vault_authority_account, &from_nft_account_info, &to_nft_account_info, 1, &[seeds])?;

        if !metadata_account_info.data_is_empty() {
            total_royalty_fee_transferred = transfer_royalty_fee(account_iter, metadata_account_info, vault_authority_account, buyer_package_token_account, &package_token, amount_after_sub_system_fee, &[seeds])?;
        }
    } else {
        if vault_account.total_royalty_fee > 0 {
            let vault_royalty_fee_owner = next_account_info(account_iter)?;
            require!(vault_royalty_fee_owner.key() == get_associated_token_address(&vault_authority_account.key(), &package_token), ErrorCode::InvalidVaultRoyaltyFeeOwner);
            let royalty_fee = package_account.royalty_fee;

            total_royalty_fee_transferred = amount_after_sub_system_fee.checked_mul(royalty_fee.into()).unwrap().checked_div(PERCENT.into()).unwrap();

            transfer_token(vault_authority_account, buyer_package_token_account, vault_royalty_fee_owner, total_royalty_fee_transferred, &[seeds])?;
        }
    }

    // transfer amount to seller
    transfer_token(
        vault_authority_account,
        &buyer_package_token_account.to_account_info(),
        &seller_package_token_account.to_account_info(),
amount_after_sub_system_fee.checked_sub(total_royalty_fee_transferred).unwrap()
,
        &[seeds]
    )?;

    let vault_account = &mut ctx.accounts.vault_account;

    vault_account.owner = order_account.buyer;
    vault_account.status = VaultStatus::Sold;

    Ok(())
}
```

However, as long as the `execute_order()` function is not executed yet, the seller can still change the vault account status at any time by using the `cancel_listing_for_sale()` function.

lib.rs

```
pub fn cancel_listing_for_sale(
  ctx: Context<CancelListingForSaleContext>
) -> Result<()> {
  msg!("DAgora Marketplace: Cancel List Item For Sale Instruction");
  let vault_account = &mut ctx.accounts.vault_account;

  vault_account.status = VaultStatus::Create;

  emit!(CancelListingEvent{});

  Ok(())
}
```

It results in the buyer creating an offer, then the seller can accept the offer and cancel the listing NFT before using the `withdraw_item()` function to withdraw the NFT from the vault account and deliver the empty vault to the buyer by executing the `execute_order()` function.

When the buyer offers an order to buy the NFTs, In the case of the vault account created from the `create_multi_items_vault()` function.

1. The seller accepts an offer to change the `order_account.status` to `OrderStatus::Accept`.

2. The seller uses the `cancel_listing_for_sale()` function to cancel the listing for sale, which changes the `vault_account.status` to `VaultStatus::Create`.
3. The seller can execute the `withdraw_item()` function to withdraw the NFT from the vault to the seller.
4. Execute the `execute_order()` function to transfer the user's tokens and change the ownership of the empty vault account to the buyer.

5.2.2. Remediation

Inspex suggests validating that the listing account has been removed or not when executing the `execute_order()` function to prevent the seller from withdrawing the NFT. Moreover, the `execute_order()` function of the program should remove the `listing_account` account after the execution is success, for example:

instructions.rs

```
#[derive(Accounts)]
#[instruction(package_token: Pubkey)]
pub struct ExecuteOrderContext<'info> {
  #[account(mut)]
  pub payer: Signer<'info>,

  #[account(
    seeds = [
      package_token.key().as_ref()
    ],
    bump = package_account.nonce,
    constraint = package_account.is_active @ErrorCode::PackageNotActiveYet,
  )]
  pub package_account: Box<Account<'info, PackageInfo>>,

  #[account(
    mut,
    constraint = vault_account.package_token == package_token
@ErrorCode::InvalidPackageAccount,
  )]
  pub vault_account: Account<'info, VaultInfo>,

  /// CHECK: Authority of Vault account
  #[account(
    seeds = [
      AUTHORITY_SEED,
      vault_account.to_account_info().key.as_ref()
    ],
    bump = vault_account.authority_nonce
  )]
  pub vault_authority_account: AccountInfo<'info>,

 #[account(
  mut,
  constraint = listing_account.vault_account == vault_account.key() @ErrorCode::InvalidVaultAccount,
  close = payer
 )]
 pub listing_account: Account<'info, ListingForSaleInfo>,

 #[account(
  constraint = order_account.vault_account == *vault_account.to_account_info().key @ErrorCode::InvalidVaultAccount,
  constraint = order_account.status == OrderStatus::Accept @ErrorCode::InvalidOrderStatus,
  seeds = [
   listing_account.to_account_info().key.as_ref(),
   order_account.buyer.as_ref(),
   &order_account.amount.to_le_bytes()
  ],
  bump = order_account.nonce,
 )]
 pub order_account: Box<Account<'info, OrderInfo>>,

 /// CHECK: This is not dangerous because we don't read or write from this account
 #[account(
  mut,
  address = get_associated_token_address(&order_account.buyer, &package_token)
 )]
 pub buyer_package_token_account: AccountInfo<'info>,

 /// CHECK: This is not dangerous because we don't read or write from this account
 #[account(
  mut,
  address = get_associated_token_address(&package_account.fee_owner, &package_token)
 )]
 pub fee_owner_token_address: AccountInfo<'info>,

 /// CHECK: This is not dangerous because we don't read or write from this account
 #[account(
  mut,
  address = get_associated_token_address(&vault_account.owner, &package_token)
 )]
 pub seller_package_token_account: AccountInfo<'info>,

 /// CHECK: We have checked address
 #[account(
  address = TOKEN_PROGRAM_ID
 )]
 pub token_program: AccountInfo<'info>,
 }
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.3. Upgradability of Solana Program

ID	IDX-003
Target	dagora_solana
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 of 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.

5.3.1. 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.

5.3.2. 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

5.4. Centralized Control of State Variable

ID	IDX-004
Target	dagora_solana
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved * The DAgora team has mitigated this issue by confirming that they will use the multisig account as an authorized party to ensure that all privilege contracts are well prepared since the multisig account's execution requires that a list of members in the authorized party must agree.

5.4.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

Each package's fee is changed through the `update_package()` function, which can be called by the admin to change the fee at any time.

lib.rs

```rust
#[access_control(verify_root(*ctx.accounts.root.key))]
pub fn update_package(
  ctx: Context<UpdatePackageContext>,
  fee_owner: Pubkey,
  is_active: bool,
  market_fee: u16,
  claim_fee: u64,
  royalty_fee: u16,
) -> Result<()> {
  msg!("DAgora Marketplace: Update Package Instruction");

  let package_account = &mut ctx.accounts.package_account;

  package_account.fee_owner = fee_owner;
  package_account.is_active = is_active;
  package_account.market_fee = market_fee;
  package_account.claim_fee = claim_fee;
  package_account.royalty_fee= royalty_fee;

  emit!(UpdatePackageEvent{
    fee_owner,
    is_active,
    market_fee,
    claim_fee,
    royalty_fee
  });

  Ok(())
}
```

5.4.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the program. However, if modifications are needed, Inspex suggests limiting the use of these functions by the following options:

- Using a multisig account controlled by multiple trusted parties to ensure that the changes of critical states are well prepared
- Implementing a community-run governance to control the use of these functions

5.5. Design Flaw in Auction Mechanism

ID	IDX-005
Target	dagora_solana
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The NFT seller will lose their NFT, which will be permanently locked in the vault account. The impact was reduced to medium after the reassessment because the NFT seller was able to withdraw the NFT from the vault account when the vault status changed to canceled due to the failed auction.The remaining impact is that the highest bidder could still revoke the ATA's permission to make any auction fail. **Likelihood:** Medium It is likely that this attack scenario will happen since the attacker can simply bid any amount to the auction NFT. When the auction is about to end, the attacker can revoke the account delegation to the vault authority, resulting in being unable to transfer the bidder's token to the seller.
Status	Acknowledged The DAgora team has partially resolved this issue by implementing the bidder delegate validation, which checks if the bidder has sufficient tokens. The remaining concern is that when the auction period is over, the highest bidder can still revoke the ATA permission that was granted for the `vault_authority_account` account, and the `execute_order()` function will fail to transfer the token. This could cause any auction to fail.

5.5.1. Description

The DAgora marketplace allows sellers to place their NFTs up for auction in a period of time. After that, any user can place a bid through the `place_a_bid()` function, and the NFT will be transferred to the user who has the highest bid amount when the auction ends by the platform admin calling the `end_bid()` and `execute_order()` functions respectively.

Basically, when the user places a bid for an auction via the `place_a_bid()` function, the buyer's ATA (associated token account) will approve the `vault_authority_account` account in order to allow the `vault_authority_account` account to transfer the buyer's token to the seller, at line 325.

lib.rs

```
295 pub fn place_a_bid(
296     ctx: Context<PlaceABidContext>,
```

```rust
  _package_token: Pubkey,
  amount: u64,
) -> Result<()> {
  msg!("DAgora Marketplace: Place A Bid Instruction");

  let buyer = &ctx.accounts.buyer;
  let listing_account = &ctx.accounts.listing_account;
  let vault_authority_account = &ctx.accounts.vault_authority_account;
  let order_account = &mut ctx.accounts.order_account;
  let buyer_package_token_account = &ctx.accounts.buyer_package_token_account;

  order_account.buyer = *buyer.to_account_info().key;
  order_account.amount = amount;

  let current_time = Clock::get().unwrap().unix_timestamp;

  if listing_account.start_time > 0 {
    require!(current_time >= listing_account.start_time.try_into().unwrap(), ErrorCode::InvalidAuctionTime);
  }

  if listing_account.end_time > 0 {
    require!(current_time <= listing_account.end_time.try_into().unwrap(), ErrorCode::InvalidAuctionTime);
  }

  if amount >= listing_account.buy_immediate_amount {
    order_account.status = OrderStatus::Accept;
  }

  approve_token(&buyer.to_account_info(), &buyer_package_token_account.to_account_info(), &vault_authority_account.to_account_info(), amount, &[])?;

  emit!(PlaceABidEvent{
    amount
  });

  Ok(())
}
```

When the auction period is over, the platform's owner will call the `end_bid()` function to end an auction and change the`order_account.status` to `OrderStatus::Accept` which is shown below in line 346.

lib.rs

```rust
335 pub fn end_bid(
336   ctx: Context<EndBidContext>,
337 ) -> Result<()> {
338   msg!("DAgora Marketplace: End Bid Instruction");
339 
340   let order_account = &mut ctx.accounts.order_account;
341   let listing_account = &ctx.accounts.listing_account;
342 
343   let current_time = Clock::get().unwrap().unix_timestamp;
344 
345   if current_time > listing_account.end_time.try_into().unwrap() &&
    order_account.amount > listing_account.start_amount {
346     order_account.status = OrderStatus::Accept;
347   }
348 
349   emit!(EndBidEvent{
350   });
351 
352   Ok(())
353 }
```

Before the auction period is over, the highest bidder can revoke the ATA (associated token account) permission that was granted for the `vault_authority_account` account, the `execute_order()` function will fail to transfer the token.

In addition, if the token in the wallet of highest bidder is not enough, the `execute_order()` function will also fail to transfer the token as shown in lines 478 - 484 and 519 - 525.

lib.rs

```
455 pub fn execute_order<'info>(
456   ctx: Context<'_, '_, '_, 'info, ExecuteOrderContext<'info>>,
457   package_token: Pubkey,
458 ) -> Result<()>{
459   msg!("DAgora Marketplace: Execute Order Instruction");
460 
461   let package_account = &ctx.accounts.package_account;
462   let order_account = &ctx.accounts.order_account;
463 
464   let vault_account = &ctx.accounts.vault_account;
465   let vault_account_key = vault_account.key();
466 
467   let vault_authority_account = &ctx.accounts.vault_authority_account;
468 
469   let buyer_package_token_account = &ctx.accounts.buyer_packag_token_account;
470   let fee_owner_token_address = &ctx.accounts.fee_owner_token_address;
471   let seller_package_token_account =
    &ctx.accounts.seller_package_token_account;
472 
```

```
  let seeds: &[&[_]] = &[AUTHORITY_SEED, &vault_account_key.as_ref(), &[vault_account.authority_nonce]];

  let (system_fee, amount_after_sub_system_fee) = order_account.split_amount(package_account.market_fee, package_account.claim_fee);

  // transfer system fee
  transfer_token(
    vault_authority_account,
    &buyer_package_token_account.to_account_info(),
    &fee_owner_token_address.to_account_info(),
    system_fee,
    &[seeds]
  )?;

  let account_iter = &mut ctx.remaining_accounts.iter();

  let mut total_royalty_fee_transferred: u64 = 0;

  let vault_account = &ctx.accounts.vault_account;

  if vault_account.vault_type == VaultType::SingleItem {
    let from_nft_account_info = next_account_info(account_iter)?;
    let to_nft_account_info = next_account_info(account_iter)?;
    let metadata_account_info = next_account_info(account_iter)?;

    require!(from_nft_account_info.key() == get_associated_token_address(&vault_account.owner, &vault_account.nft_mints[0]), ErrorCode::InvalidSellerNftTokenAccount);
    require!(to_nft_account_info.key() == get_associated_token_address(&order_account.buyer, &vault_account.nft_mints[0]), ErrorCode::InvalidBuyerNftTokenAccount);
    require!(metadata_account_info.key() == find_metadata_account(&vault_account.nft_mints[0]).0, ErrorCode::InvalidMetadataAccount);

    transfer_token(vault_authority_account, &from_nft_account_info, &to_nft_account_info, 1, &[seeds])?;

    if !metadata_account_info.data_is_empty() {
      total_royalty_fee_transferred = transfer_royalty_fee(account_iter, metadata_account_info, vault_authority_account, buyer_package_token_account, &package_token, amount_after_sub_system_fee, &[seeds])?;
    }
  } else {
    if vault_account.total_royalty_fee > 0 {
      let vault_royalty_fee_owner = next_account_info(account_iter)?;
      require!(vault_royalty_fee_owner.key() == get_associated_token_address(&vault_authority_account.key(), &package_token), ErrorCode::InvalidVaultRoyaltyFeeOwner);
      let royalty_fee = package_account.royalty_fee;

      total_royalty_fee_transferred = amount_after_sub_system_fee.checked_mul(royalty_fee.into()).unwrap().checked_div(PERCENT.into()).unwrap();

      transfer_token(vault_authority_account, buyer_package_token_account, vault_royalty_fee_owner, total_royalty_fee_transferred, &[seeds])?;
    }
  }

  // transfer amount to seller
  transfer_token(
    vault_authority_account,
    &buyer_package_token_account.to_account_info(),
    &seller_package_token_account.to_account_info(),
    amount_after_sub_system_fee.checked_sub(total_royalty_fee_transferred).unwrap(),
    &[seeds]
  )?;

  let vault_account = &mut ctx.accounts.vault_account;

  vault_account.owner = order_account.buyer;
  vault_account.status = VaultStatus::Sold;

  Ok(())
}
```

Moreover, when the auction ends, the seller will not be able to cancel the auction due to the amount state of the `bid_account` account being updated as shown in line 386.

lib.rs

```
368 pub fn cancel_listing_for_auction(
369   ctx: Context<CancelListingForAuctionContext>
370 ) -> Result<()> {
371   msg!("DAgora Marketplace: Cancel List Item For Sale Instruction");
372
373   let listing_account = &ctx.accounts.listing_account;
374   let bid_account = &ctx.accounts.bid_account;
375   let vault_account = &mut ctx.accounts.vault_account;
376
377   let current_time = Clock::get().unwrap().unix_timestamp;
```

```

  if vault_account.status == VaultStatus::Sold {
    return Ok(());
  }

  vault_account.status = VaultStatus::Create;

  if current_time > listing_account.end_time.try_into().unwrap() {
    require!(bid_account.amount == listing_account.start_amount,
ErrorCode::InvalidAuctionTime);
  } else {
    require!(current_time < listing_account.start_time.try_into().unwrap(),
ErrorCode::InvalidAuctionTime);
  }

  emit!(CancelListingEvent{});

  Ok(())
}
```

As a result, the seller will lose their NFT, which will be permanently locked in the vault account.

5.5.2. Remediation

Inspex suggests implementing the mechanism to ensure that the seller can withdraw the NFT. For example:

- Implementing the bidding wallet to ensure that the bidder will have enough tokens to bid.
- The `end_bid()` function should ensure that the bidder has sufficient tokens. If the bidding fails, the owner will be able to transfer an NFT from the vault.

5.6. Unbound Configuration Parameter

ID	IDX-006
Target	dagora_solana
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The token package's fee (payment token) can be set up to 100%, which means that the entire token amount from buying NFT is paid to the market fee, leaving the NFT seller to receive nothing. **Likelihood:** Low The fee attribute of each token package (payment token) can only be set by the owner of the DAgora marketplace program whose addresses are whitelisted in the `ROOT_KEYS` variable.
Status	Acknowledged The DAgora team has partially resolved this issue by implementing the bound configuration for the suggested parameters except the `claimFee` according to the business design.

5.6.1. Description

The DAgora marketplace is the middleman platform for exchanging NFT that allows users to buy and sell the NFT and charges exchange fees.

The buyer will pay a platform and a creator fee for each NFT trade on the DAgora marketplace, where the fee is different for each package generated by the platform owner. That is shown in the source code by line 475.

lib.rs

```
455 pub fn execute_order<'info>(
456   ctx: Context<'_, '_, '_, 'info, ExecuteOrderContext<'info>>,
457   package_token: Pubkey,
458 ) -> Result<()>{
459   msg!("DAgora Marketplace: Execute Order Instruction");
460
461   let package_account = &ctx.accounts.package_account;
462   let order_account = &ctx.accounts.order_account;
463
464   let vault_account = &ctx.accounts.vault_account;
465   let vault_account_key = vault_account.key();
```

```

  let vault_authority_account = &ctx.accounts.vault_authority_account;

  let buyer_package_token_account = &ctx.accounts.buyer_package_token_account;
  let fee_owner_token_address = &ctx.accounts.fee_owner_token_address;
  let seller_package_token_account = &ctx.accounts.seller_package_token_account;

  let seeds: &[&[_]] = &[AUTHORITY_SEED, &vault_account_key.as_ref(), &[vault_account.authority_nonce]];

  let (system_fee, amount_after_sub_system_fee) = order_account.split_amount(package_account.market_fee, package_account.claim_fee);

  // transfer system fee
  transfer_token(
    vault_authority_account,
    &buyer_package_token_account.to_account_info(),
    &fee_owner_token_address.to_account_info(),
    system_fee,
    &[seeds]
  )?;

  let account_iter = &mut ctx.remaining_accounts.iter();

  let mut total_royalty_fee_transferred: u64 = 0;

  let vault_account = &ctx.accounts.vault_account;

  if vault_account.vault_type == VaultType::SingleItem {
    let from_nft_account_info = next_account_info(account_iter)?;
    let to_nft_account_info = next_account_info(account_iter)?;
    let metadata_account_info = next_account_info(account_iter)?;

    require!(from_nft_account_info.key() == get_associated_token_address(&vault_account.owner, &vault_account.nft_mints[0]), ErrorCode::InvalidSellerNftTokenAccount);
    require!(to_nft_account_info.key() == get_associated_token_address(&order_account.buyer, &vault_account.nft_mints[0]), ErrorCode::InvalidBuyerNftTokenAccount);
    require!(metadata_account_info.key() == find_metadata_account(&vault_account.nft_mints[0]).0, ErrorCode::InvalidMetadataAccount);

    transfer_token(vault_authority_account, &from_nft_account_info, &to_nft_account_info, 1, &[seeds])?;

    if !metadata_account_info.data_is_empty() {
      total_royalty_fee_transferred = transfer_royalty_fee(account_iter, metadata_account_info, vault_authority_account, buyer_package_token_account, &package_token, amount_after_sub_system_fee, &[seeds])?;
    }
  } else {
    if vault_account.total_royalty_fee > 0 {
      let vault_royalty_fee_owner = next_account_info(account_iter)?;
      require!(vault_royalty_fee_owner.key() == get_associated_token_address(&vault_authority_account.key(), &package_token), ErrorCode::InvalidVaultRoyaltyFeeOwner);
      let royalty_fee = package_account.royalty_fee;

      total_royalty_fee_transferred = amount_after_sub_system_fee.checked_mul(royalty_fee.into()).unwrap().checked_div(PERCENT.into()).unwrap();

      transfer_token(vault_authority_account, buyer_package_token_account, vault_royalty_fee_owner, total_royalty_fee_transferred, &[seeds])?;
    }
  }

  // transfer amount to seller
  transfer_token(
    vault_authority_account,
    &buyer_package_token_account.to_account_info(),
    &seller_package_token_account.to_account_info(),
    amount_after_sub_system_fee.checked_sub(total_royalty_fee_transferred).unwrap(),
    &[seeds]
  )?;

  let vault_account = &mut ctx.accounts.vault_account;

  vault_account.owner = order_account.buyer;
  vault_account.status = VaultStatus::Sold;

  Ok(())
}
```

Since the `update_package()` function does not have an update boundary, the platform owner can change the fee value up to the `PERCENT` constant. Therefore, the seller would receive nothing because the platform fee might be as high as 100% of the NFT price.

lib.rs

```
#[access_control(verify_root(*ctx.accounts.root.key))]
pub fn update_package(
  ctx: Context<UpdatePackageContext>,
  fee_owner: Pubkey,
  is_active: bool,
  market_fee: u16,
  claim_fee: u64,
  royalty_fee: u16,
) -> Result<()> {
  msg!("DAgora Marketplace: Update Package Instruction");

  let package_account = &mut ctx.accounts.package_account;

  package_account.fee_owner = fee_owner;
  package_account.is_active = is_active;
  package_account.market_fee = market_fee;
  package_account.claim_fee = claim_fee;
  package_account.royalty_fee= royalty_fee;

  emit!(UpdatePackageEvent{
    fee_owner,
    is_active,
    market_fee,
    claim_fee,
    royalty_fee
  });

  Ok(())
}
```

5.6.2. Remediation

Inspex suggests adding input validation to ensure that the input fee does not exceed the possible maximum fee cap by declaring the constant variable max fee for an individual state, which the potential value will follow by the DAgora business model. For example,

constants.rs

```
pub const ROOT_KEYS: &[&str] = &["GnzQDYm2gvwZ8wRVmuwVAeHx5T44ovC735vDgSNhumzQ"];
pub const TOKEN_PROGRAM_ID: Pubkey = Pubkey::new_from_array([6, 221, 246, 225, 215, 101, 161, 147, 217, 203, 225, 70, 206, 235, 121, 172, 28, 180, 133, 237, 95, 91, 55, 145, 58, 140, 245, 133, 126, 255, 0, 169]);
pub const TOKEN_METADATA_PROGRAM_ID: Pubkey = Pubkey::new_from_array([6, 221, 246, 225, 215, 101, 161, 147, 217, 203, 225, 70, 206, 235, 121, 172, 28, 180, 133, 237, 95, 91, 55, 145, 58, 140, 245, 133, 126, 255, 0, 169]);

pub const AUTHORITY_SEED: &[u8] = b"authority";

pub const PERCENT: u16 = 10000;

pub const TOTAL_ROYALTY_FEE: u16 = 2000;

pub const ROYALTY_FEE_CAP: u16 = 2000;
pub const MARKET_FEE_CAP: u16 = 500;
pub const CLAIM_FEE_CAP: u16 = 100;
```

lib.rs

```
#[access_control(verify_root(*ctx.accounts.root.key))]
pub fn update_package(
  ctx: Context<UpdatePackageContext>,
  fee_owner: Pubkey,
  is_active: bool,
  market_fee: u16,
  claim_fee: u64,
  royalty_fee: u16,
) -> Result<()> {
  require!(market_fee <= MARKET_FEE_CAP, ErrorCode::InvalidFeeCap);
  require!(claim_fee <= CLAIM_FEE_CAP, ErrorCode::InvalidFeeCap);
  require!(royalty_fee <= ROYALTY_FEE_CAP, ErrorCode::InvalidFeeCap);

  msg!("DAgora Marketplace: Update Package Instruction");

  let package_account = &mut ctx.accounts.package_account;

  package_account.fee_owner = fee_owner;
  package_account.is_active = is_active;
  package_account.market_fee = market_fee;
  package_account.claim_fee = claim_fee;
  package_account.royalty_fee= royalty_fee;

  emit!(UpdatePackageEvent{
    fee_owner,
    is_active,
    market_fee,
    claim_fee,
    royalty_fee
  });

  Ok(())
}
```

5.7. Smart Contract with Unpublished Source Code

ID	IDX-007
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.

5.7.1. 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.

5.7.2. 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