Marketplace (Solana)

Smart Contract Audit Report
Prepared for DAgora

Date Issued:	Aug 21, 2023
Project ID:	AUDIT2022050
Version:	v1.0
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022050
Version	v1.0
Client	DAgora
Project	Marketplace (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 Marketplace (Solana) smart contracts on Oct 26, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Marketplace (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 1 critical, 2 high, 2 low, 1 very low-severity issues. With the project team's prompt response 1 critical, 2 high, 1 very low-severity issues were resolved or mitigated in the reassessment, while 2 low-severity issues were acknowledged by the team. Therefore, Inspex trusts that Marketplace (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.

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, which can accept many tokens at the same time. The platform also provides the royalty fee for the NFT creators, helping them to gain their revenue.

Scope Information:

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

Audit Information:

Audit Method	Whitebox
Audit Date	Oct 26, 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_solana	0129cca31555fc35de1d0c9ec686a7bb8e54e314bc9a164d7900caf5bd80f937

Reassessment

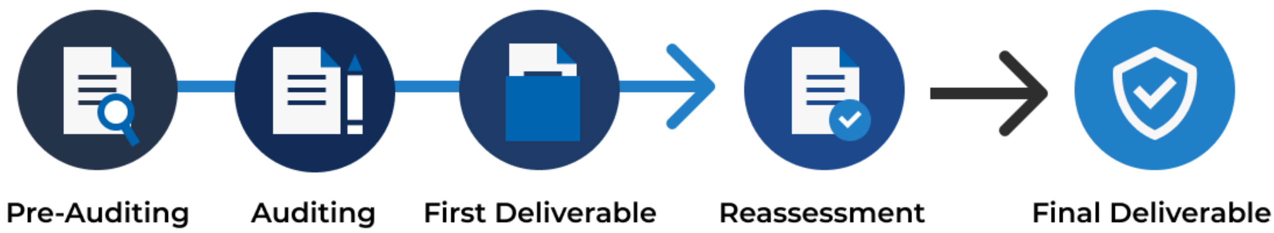
Contract	Bytecode SHA256 Hash
dagora_solana	3da31cedc2b58ddcac2b8928bc9e6a7f9c838143bc0d69354005e57f132c64ad

As the Coin98 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**.

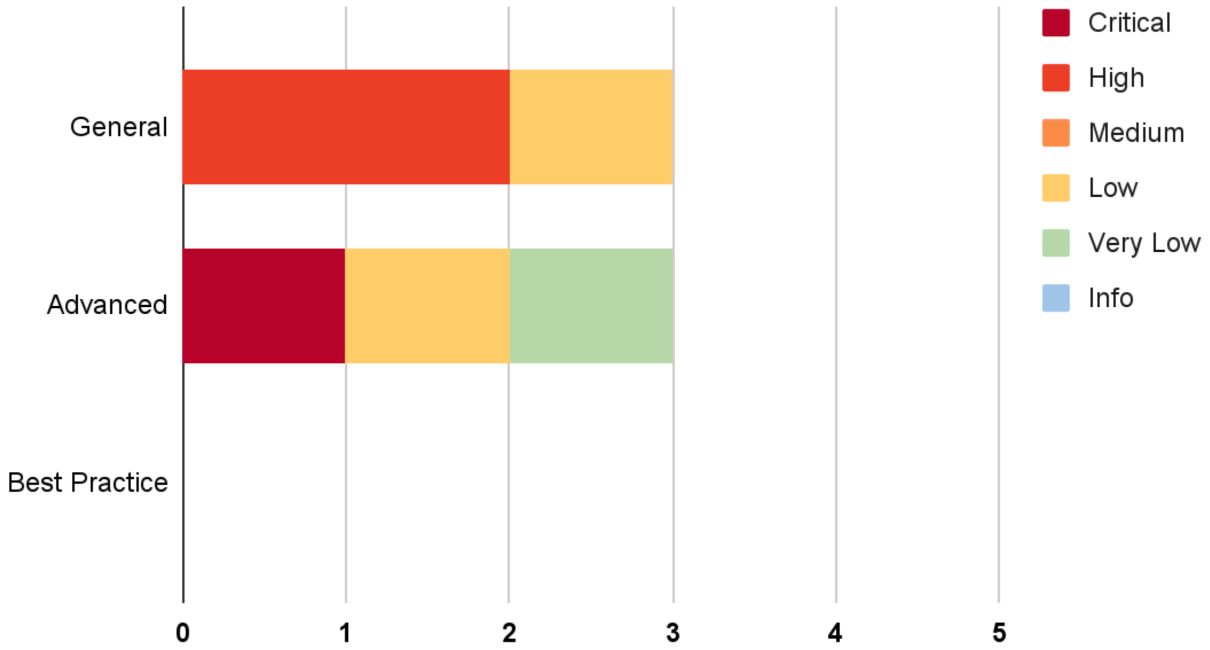
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

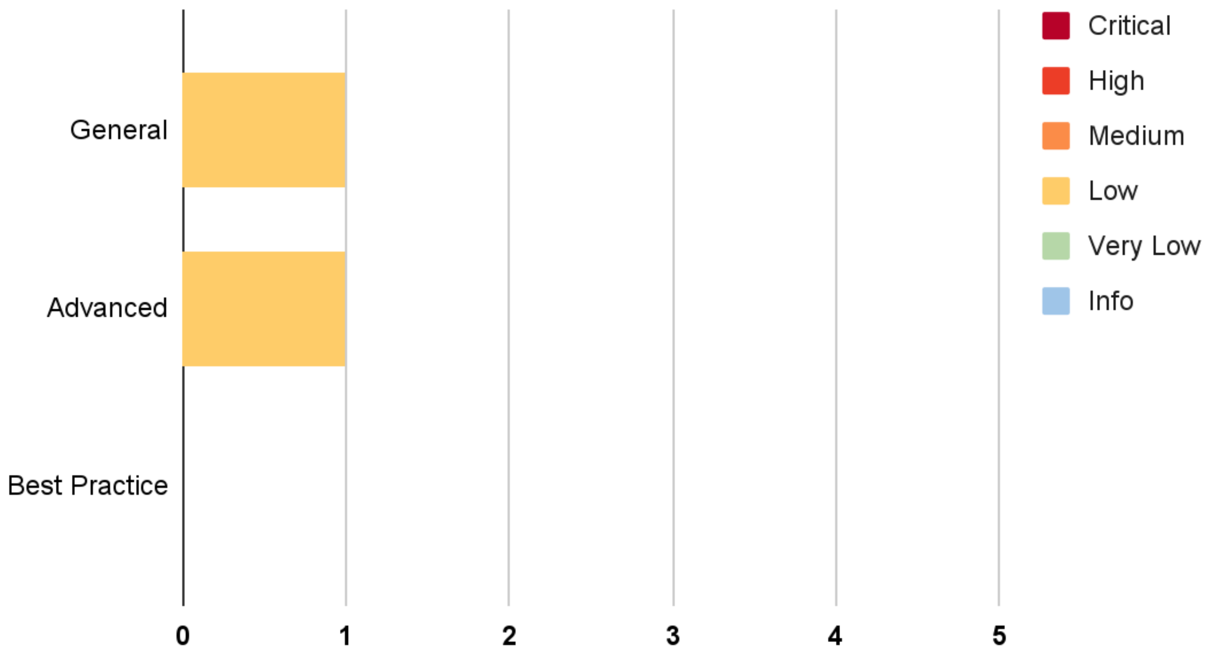
4. Summary of Findings

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	Insufficient package_token Validation	Advanced	Critical	Resolved
IDX-002	Upgradability of Solana Program	General	High	Resolved *
IDX-003	Centralized Control of State Variable	General	High	Resolved *
IDX-004	Design Flaw in Auction Mechanism	Advanced	Low	Acknowledged
IDX-005	Smart Contract with Unpublished Source Code	General	Low	Acknowledged
IDX-006	Unable to Offer same Amount with Different Token	Advanced	Very Low	Resolved

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

5. Detailed Findings Information

5.1. Insufficient package_token Validation

ID	IDX-001
Target	dagora_solana
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** The `execute_order()` function does not validate that the `package_token` parameter is the same package token that was accepted in the `create_offer()` function. The attacker can buy the NFT vault at a cheaper price than the seller listing their NFT vault for sale. **Likelihood: High** This issue is likely to be exploited since there is no mechanism to validate that the `package_token` token is the same that the seller wants to sell.
Status	Resolved The DAgora team has resolved this issue as suggested.

5.1.1. Description

When the seller lists an NFT vault for sale, the array of the accepted tokens and amounts will be stored in the `listing_account` account, as shown below in lines 258–259.

lib.rs

```
242 pub fn listing_for_sale(
243   ctx: Context<ListingForSaleContext>,
244   _listing_account_path: Vec<u8>,
245   package_tokens: Vec<Pubkey>,
246   amounts: Vec<u64>,
247 ) -> Result<()> {
248   msg!("DAgora Marketplace: List For Sale Instruction");
249
250   let seller = &ctx.accounts.seller;
251   let vault_account = &mut ctx.accounts.vault_account;
252   let listing_account = &mut ctx.accounts.listing_account;
253
254   vault_account.status = VaultStatus::OnListing;
255
256   listing_account.vault_account = *vault_account.to_account_info().key;
257
```

```
  listing_account.package_tokens = package_tokens.clone();
  listing_account.amounts = amounts.clone();

  emit!(ListingForSaleEvent{
    seller: seller.key(),
    vault: vault_account.key(),
    listing: listing_account.key(),
    package_tokens,
    amounts
  });

  Ok(())
}
```

For the buyer, the `create_offer()` function is used to offer or buy an NFT vault. When the offer price is higher or equal to the `listing_account.amounts[package_index]` state, the buyer will immediately buy the NFT vault as shown below in lines 513-515.

lib.rs

```
490 pub fn create_offer(
491   ctx: Context<CreateOfferContext>,
492   package_token: Pubkey,
493   amount: u64,
494 ) -> Result<()> {
495   msg!("DAgora Marketplace: Create Offer Instruction");
496   let buyer = &ctx.accounts.buyer;
497   let order_account = &mut ctx.accounts.order_account;
498
499   let vault_account = &mut ctx.accounts.vault_account;
500   let vault_authority_account = &ctx.accounts.vault_authority_account;
501   let listing_account = &ctx.accounts.listing_account;
502   let buyer_package_token_account = &ctx.accounts.buyer_package_token_account;
503
504   order_account.order_type = OrderType::Offer;
505   order_account.nonce = *ctx.bumps.get("order_account").unwrap();
506   order_account.vault_account = listing_account.vault_account;
507   order_account.buyer = *buyer.to_account_info().key;
508   order_account.listing_account = *listing_account.to_account_info().key;
509   order_account.amount = amount;
510
511   let package_index = listing_account.package_tokens.iter().position(|package|
package == &package_token).unwrap();
512
513   if amount >= listing_account.amounts[package_index] {
514     order_account.status = OrderStatus::Accept;
515     vault_account.status = VaultStatus::WaitingForExecute;
```

```
  } 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{
    listing: listing_account.key(),
    buyer: buyer.key(),
    order: order_account.key(),
    amount
  });

  Ok(())
}
```

After calling the `create_offer()` function, the buyer calls the `execute_order()` function to process the buy action with the `package_token` parameter.

lib.rs

```
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 = &mut 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]];

  // validate buyer delegate amount
  if get_token_delegate_amount(buyer_package_token_account,
&vault_authority_account.key()) < order_account.amount {
```

```
    vault_account.status = VaultStatus::Cancel;
    return Ok(())
  }

  vault_account.sold_by_package = package_token;

  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;

  emit!(ExecuteOrderEvent{
    package: package_account.key(),
    vault: vault_account_key,
    order: order_account.key()
  });

  Ok(())
}
```

However, the `package_token` parameter of the `execute_order()` function is controlled by the buyer without any validation. Resulting in the buyer being able to buy the NFT vault at a cheaper price in the following scenario:

1. The seller lists the NFT vault for sale for 3 $SOL (let's say 1 $SOL is 30 $USDC).
2. The malicious buyer creates an offer for 3 $SOL to immediately buy the NFT vault.
3. The malicious buyer called the `execute_order()` function, which used the `order_account` from (2) but passed the `package_token` of $USDC instead of $SOL.
4. The malicious buyer pays only 3 $USDC for the NFT vault.

5.1.2. Remediation

Inspex suggests validating the provided `package_token` address to the `execute_order()` function is the same as the token address that is accepted via the `create_offer()` function.

For example, assign the accepted package token to the `order_account` account in the `create_offer()` function as shown in line 510.

lib.rs

```
490 pub fn create_offer(
491   ctx: Context<CreateOfferContext>,
492   package_token: Pubkey,
493   amount: u64,
494 ) -> Result<()> {
495   msg!("DAgora Marketplace: Create Offer Instruction");
496   let buyer = &ctx.accounts.buyer;
497   let order_account = &mut ctx.accounts.order_account;
498
499   let vault_account = &mut ctx.accounts.vault_account;
500   let vault_authority_account = &ctx.accounts.vault_authority_account;
501   let listing_account = &ctx.accounts.listing_account;
502   let buyer_package_token_account = &ctx.accounts.buyer_package_token_account;
503
504   order_account.order_type = OrderType::Offer;
505   order_account.nonce = *ctx.bumps.get("order_account").unwrap();
506   order_account.vault_account = listing_account.vault_account;
507   order_account.buyer = *buyer.to_account_info().key;
508   order_account.listing_account = *listing_account.to_account_info().key;
509   order_account.amount = amount;
510   order_account.package_token = package_token;
511
512   let package_index = listing_account.package_tokens.iter().position(|package|
package == &package_token).unwrap();
513
514   if amount >= listing_account.amounts[package_index] {
515     order_account.status = OrderStatus::Accept;
516     vault_account.status = VaultStatus::WaitingForExecute;
517   } else {
518     order_account.status = OrderStatus::Created;
519   }
520
```

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

  emit!(CreateOfferEvent{
    listing: listing_account.key(),
    buyer: buyer.key(),
    order: order_account.key(),
    amount
  });

  Ok(())
}
```

Then, allow only the `package_token` address equal to the `order_account.package_token` address at line 634.

instructions.rs

```
598 #[derive(Accounts)]
599 #[instruction(package_token: Pubkey)]
600 pub struct ExecuteOrderContext<'info> {
601   #[account(mut)]
602   pub signer: Signer<'info>,
603
604   #[account(
605     seeds = [
606       package_token.key().as_ref()
607     ],
608     bump = package_account.nonce,
609     constraint = package_account.is_active @ErrorCode::PackageNotActiveYet,
610   )]
611   pub package_account: Box<Account<'info, PackageInfo>>,
612
613   #[account(
614     mut,
615     constraint = vault_account.status == VaultStatus::WaitingForExecute @ErrorCode::InvalidVaultStatus,
616   )]
617   pub vault_account: Account<'info, VaultInfo>,
618
619   /// CHECK: Authority of Vault account
620   #[account(
621     seeds = [
622       AUTHORITY_SEED,
623       vault_account.to_account_info().key.as_ref()
624     ],
625     bump = vault_account.authority_nonce
```

```
  )]
  pub vault_authority_account: AccountInfo<'info>,

  /// CHECK: close account after execute instruction
  #[account(
    mut,
    constraint = order_account.vault_account == *vault_account.to_account_info().key @ErrorCode::InvalidVaultAccount,
    constraint = order_account.status == OrderStatus::Accept @ErrorCode::InvalidOrderStatus,
    constraint = order_account.package_token == package_token @ErrorCode::InvalidPackageToken,
    close = signer
  )]
  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>,
}
```

5.2. Upgradability of Solana Program

ID	IDX-002
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 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.

5.2.1. 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.

5.2.2. 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

5.3. Centralized Control of State Variable

ID	IDX-003
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 program owner.
Status	Resolved * The DAgora team has mitigated this issue by confirming that the critical program state modification authority will be a multisig account controlled by multiple trusted parties.

5.3.1. 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 some 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;
```

```

  require!(market_fee <= MARKET_FEE_CAP, ErrorCode::InvalidFeeCap);
  require!(royalty_fee <= ROYALTY_FEE_CAP, ErrorCode::InvalidFeeCap);

  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{
    package: package_account.key(),
    fee_owner,
    is_active,
    market_fee,
    claim_fee,
    royalty_fee
  });

  Ok(())
}
```

5.3.2. 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

5.4. Design Flaw in Auction Mechanism

ID	IDX-004
Target	dagora_solana
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low This issue possibly damages the platform's reputation because an attacker can disrupt the auction, which is a bad experience for both the seller and the bidder. **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 acknowledged this issue since the program is not intended to hold the user's token in accordance with the business design.

5.4.1. 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 413.

lib.rs

```
381 pub fn place_a_bid(
382   ctx: Context<PlaceABidContext>,
383   _package_token: Pubkey,
384   amount: u64,
385 ) -> Result<()> {
386   msg!("DAgora Marketplace: Place A Bid Instruction");
387 
388   let buyer = &ctx.accounts.buyer;
389   let listing_account = &ctx.accounts.listing_account;
```

```
  let vault_account = &mut ctx.accounts.vault_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;
    vault_account.status = VaultStatus::WaitingForExecute;
  }

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

  emit!(PlaceABidEvent{
    bidder: buyer.key(),
    listing: listing_account.key(),
    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 437.

lib.rs

```
424 #[access_control(verify_root(*ctx.accounts.root.key))]
425 pub fn end_bid(
426   ctx: Context<EndBidContext>,
427 ) -> Result<()> {
428   msg!("DAgora Marketplace: End Bid Instruction");
```

```

  let order_account = &mut ctx.accounts.order_account;
  let vault_account = &mut ctx.accounts.vault_account;
  let listing_account = &ctx.accounts.listing_account;

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

  if current_time > listing_account.end_time.try_into().unwrap() &&
order_account.amount > listing_account.start_amount {
    order_account.status = OrderStatus::Accept;
    vault_account.status = VaultStatus::WaitingForExecute;
  }

  emit!(EndBidEvent{
    listing: listing_account.key()
  });

  Ok(())
}
```

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 the highest bidder is not enough, the sale will be canceled, as shown in lines 583-586.

lib.rs

```
562 pub fn execute_order<'info>(
563   ctx: Context<'_, '_, '_, 'info, ExecuteOrderContext<'info>>,
564   package_token: Pubkey,
565 ) -> Result<()>{
566   msg!("DAgora Marketplace: Execute Order Instruction");
567
568   let package_account = &ctx.accounts.package_account;
569   let order_account = &ctx.accounts.order_account;
570
571   let vault_account = &mut ctx.accounts.vault_account;
572   let vault_account_key = vault_account.key();
573
574   let vault_authority_account = &ctx.accounts.vault_authority_account;
575
576   let buyer_package_token_account = &ctx.accounts.buyer_package_token_account;
577   let fee_owner_token_address = &ctx.accounts.fee_owner_token_address;
578   let seller_package_token_account =
&ctx.accounts.seller_package_token_account;
```

```

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

  // validate buyer delegate amount
  if get_token_delegate_amount(buyer_package_token_account,
&vault_authority_account.key()) < order_account.amount {
    vault_account.status = VaultStatus::Cancel;
    return Ok(())
  }

  vault_account.sold_by_package = package_token;

  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_di
v(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;

  emit!(ExecuteOrderEvent{
    package: package_account.key(),
    vault: vault_account_key,
    order: order_account.key()
  });

```

```
  Ok(())
}
```

It results in the attacker can disrupt the platform auction by bidding on all of the auctions and removing the delegate token authority before the auction ends. However, the seller will be able to withdraw their NFT after the auction ends.

5.4.2. Remediation

Inspex suggests implementing the mechanism to ensure that the bidder has enough tokens to join the auction and cannot cancel the bidding. For example, implementing the bidding wallet to ensure that the bidder will have enough tokens to pay after bidding.

5.5. Smart Contract with Unpublished Source Code

ID	IDX-005
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.

5.5.1. 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.

5.5.2. 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.

5.6. Unable to Offer Same Amount with Different Token

ID	IDX-006
Target	dagora_solana
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low The buyer cannot offer to buy with a different token for the same amount. However, it is still possible to create an offer with another price. **Likelihood:** Low It is unlikely that the buyer will offer the amount with the same previous value in the difference token.
Status	**Resolved** The DAgora team has resolved this issue as suggested.

5.6.1. Description

The DAgora Marketplace allows users to buy and sell NFTs in various tokens. The seller can define their package token by selecting which token they need to accept from the buyer.

Due to the source code that creates the `order_account` account inside the `CreateOfferContext` context in the `create_offer()` function, it is possible to create an `order_account` account that duplicates the seed usage, making it unable to create the same offer amount in the other token that is shown at lines 526-528.

instructions.rs

```
487 #[derive(Accounts)]
488 #[instruction(package_token: Pubkey, amount: u64)]
489 pub struct CreateOfferContext<'info> {
490   #[account(mut)]
491   pub buyer: Signer<'info>,
492
493   #[account(
494     seeds = [
495       package_token.key().as_ref()
496     ],
497     bump = package_account.nonce,
498     constraint = package_account.is_active @ErrorCode::PackageNotActiveYet
499   )]
500   pub package_account: Account<'info, PackageInfo>,
501
```

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

  #[account(
    init,
    seeds = [
      listing_account.to_account_info().key.as_ref(),
      buyer.to_account_info().key.as_ref(),
      &amount.to_le_bytes()
    ],
    bump,
    space = 8 + OrderInfo::LEN,
    payer = buyer,
  )]
  pub order_account: 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(&buyer.key(), &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(
```

```
545     address = TOKEN_PROGRAM_ID
546   )]
547   pub token_program: AccountInfo<'info>,
548 
549   pub system_program: Program<'info, System>,
550 }
```

For example, if the seller sells an NFT with the $USDC and $USDT, the buyer will not submit an offer with the same amount according to the following scenario.

Scenario

1. The seller listed the NFT vault for 100 $USDC and 100 $USDT.
2. The buyer makes an offer to the NFT vault for 99 $USDC.
3. The buyer makes an offer to the NFT vault again for 99 $USDT, but the transaction fails because the `order_account` is creating the same account address as (2).

5.6.2. Remediation

Inspex suggests including the `package_token` address as the seed in order to create the unique `order_account` account for each offer, for example, at line 527.

instructions.rs

```
487 #[derive(Accounts)]
488 #[instruction(package_token: Pubkey, amount: u64)]
489 pub struct CreateOfferContext<'info> {
490   #[account(mut)]
491   pub buyer: Signer<'info>,
492 
493   #[account(
494     seeds = [
495       package_token.key().as_ref()
496     ],
497     bump = package_account.nonce,
498     constraint = package_account.is_active @ErrorCode::PackageNotActiveYet
499   )]
500   pub package_account: Account<'info, PackageInfo>,
501 
502   #[account(
503     mut,
504     constraint = vault_account.status == VaultStatus::OnListing
@ErrorCode::InvalidVaultStatus,
505   )]
506   pub vault_account: Account<'info, VaultInfo>,
507 
508   /// CHECK: Authority of Vault account
```

```
509   #[account(
510     seeds = [
511       AUTHORITY_SEED,
512       vault_account.to_account_info().key.as_ref()
513     ],
514     bump = vault_account.authority_nonce
515   )]
516   pub vault_authority_account: AccountInfo<'info>,
517
518   #[account(
519     constraint = listing_account.vault_account == vault_account.key() @ErrorCode::InvalidVaultAccount,
520   )]
521   pub listing_account: Account<'info, ListingForSaleInfo>,
522
523   #[account(
524     init,
525     seeds = [
526       listing_account.to_account_info().key.as_ref(),
527       package_token.key().as_ref(),
528       buyer.to_account_info().key.as_ref(),
529       &amount.to_le_bytes()
530     ],
531     bump,
532     space = 8 + OrderInfo::LEN,
533     payer = buyer,
534   )]
535   pub order_account: Account<'info, OrderInfo>,
536
537   /// CHECK: This is not dangerous because we don't read or write from this account
538   #[account(
539     mut,
540     address = get_associated_token_address(&buyer.key(), &package_token)
541   )]
542   pub buyer_package_token_account: AccountInfo<'info>,
543
544   /// CHECK: This is not dangerous because we don't read or write from this account
545   #[account(
546     address = TOKEN_PROGRAM_ID
547   )]
548   pub token_program: AccountInfo<'info>,
549
550   pub system_program: Program<'info, System>,
551 }
```

Thus, adding the `package_token` to every `order_account` PDA derived, for example, as shown in line 590.

instructions.rs

```
#[derive(Accounts)]
pub struct AcceptOfferContext<'info> {
  #[account(mut)]
  pub seller: Signer<'info>,

  #[account(
    mut,
    constraint = vault_account.owner == *seller.to_account_info().key @ErrorCode::InvalidSellerAccount,
    constraint = vault_account.status == VaultStatus::OnListing @ErrorCode::InvalidVaultStatus
  )]
  pub vault_account: Account<'info, VaultInfo>,

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

  #[account(
    mut,
    seeds = [
      listing_account.to_account_info().key.as_ref(),
      order_account.package_token.as_ref(),
      order_account.buyer.as_ref(),
      &order_account.amount.to_le_bytes()
    ],
    bump = order_account.nonce,
  )]
  pub order_account: Account<'info, OrderInfo>,
}
```

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