STIX Swap Solana Security Review

Pashov Audit Group

Conducted by: defsec, FrankCastle, zhaojio

January 7th 2025 - January 12th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **STIX-Co/stix-swap-solana-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About STIX Swap Solana

STIX Swap contracts provide automated solutions for OTC trades, including escrow services for buyers and sellers and fee management. They support ERC20 tokens, native currencies, and allow for customizable trading setups and modular upgrades.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - d7bae15f6208b02da6029dfda2994d7b859b486e

fixes review commit hash - 9629f542572dceca4567a206202e6fdce8c59dcd

Scope

The following smart contracts were in scope of the audit:

- `lib`
- `error`
- `constants`
- `admin_struct`
- `mod`
- `offer`
- `whitelist`
- `admin`
- `cancel_offer`
- `create_offer`
- `mod`
- `taker_offer`

7. Executive Summary

Over the course of the security review, defsec, FrankCastle, zhaojio engaged with STIX Swap to review STIX Swap Solana. In this period of time a total of **20** issues were uncovered.

Protocol Summary

Protocol Name	STIX Swap Solana
Repository	https://github.com/STIX-Co/stix-swap-solana-contracts
Date	January 7th 2025 - January 12th 2025
Protocol Type	OTC Trading Service

Findings Count

Severity	**Amount**
Critical	3
High	4
Medium	2
Low	11
Total Findings	**20**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Premature closure of offer and whitelist accounts	Critical	Resolved
[C-02]	ATA initialization causes permanent DoS	Critical	Resolved
[C-03]	TokenAccounts#fee_wallet lacks constraints	Critical	Resolved
[H-01]	Missing fee calculation on input token in take_offer	High	Resolved
[H-02]	Lack of mint validation in create_offer allows arbitrary tokens	High	Resolved
[H-03]	The creator of the offer can attack the taker by front-running	High	Resolved
[H-04]	Permanent DOS and loss of funds due to unhandled token dust	High	Resolved
[M-01]	Whitelisted taker can DoS attack other whitelisted takers	Medium	Resolved
[M-02]	Same token swap protection missing	Medium	Resolved
[L-01]	Frontrunning the initialization of admin_config	Low	Resolved
[L-02]	Missing fee address validation in update function	Low	Resolved
[L-03]	Missing minimum trade size protection	Low	Acknowledged
[L-04]	State inconsistency due to Solana rollback	Low	Resolved

[L-05]	offer_id is not utilized in create_offer	Low	Resolved
[L-06]	Missing validation for an expected amount greater than zero	Low	Resolved
[L-07]	Missing Update of Offer Statistics in take_offer	Low	Resolved
[L-08]	cancel_offer requires the maker as a signer, preventing post-expiration cancellations	Low	Acknowledged
[L-09]	Division overflow may result in trading at the price of quote 0	Low	Resolved
[L-10]	Expiring offers without closing the offer PDA causes DoS	Low	Acknowledged
[L-11]	Unreliable event logging due to Solana log truncation	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Premature closure of `offer` and `whitelist` accounts

Severity

Impact: High

Likelihood: High

Description

In the `take_offer` function, the `close` constraint is applied to the `offer` and `whitelist` PDAs, as shown below:

```
#[account(
    mut,

         constraint = offer.status == OfferStatus::Ongoing @ SwapError::InvalidOfferSt
    seeds = [b"offer", offer.maker.as_ref()],
    bump,
    close = maker
)]
pub offer: Account<'info, Offer>,

#[account(
    mut,
    seeds = [b"whitelist", offer.maker.as_ref()],
    bump,
    constraint = whitelist.takers.contains(&taker.key
      ()) @ SwapError::TakerNotWhitelisted,
    close = maker
)]
pub whitelist: Account<'info, Whitelist>,
```

This results in the premature closure of both PDAs, even in cases of partial fulfillment of the offer. For example, if an offer specifies 100 tokens and the taker only accepts 2 tokens, the remaining 98 tokens will be permanently locked in the vault since the `offer` and `whitelist` accounts are closed, wiping their data and leaving the funds inaccessible.

Impact

- Loss of data related to the offer and whitelist due to closure.
- Permanent loss of remaining tokens in the vault, leading to a critical loss of user funds.

Recommendations

1. **Remove the `close` constraint** from both the `offer` and `whitelist` accounts in the `take_offer` function.

2. Introduce a `completed` flag in the `offer` and `whitelist` accounts to indicate whether the offer has been fully fulfilled. For example:

```
offer.completed = true;
```

3. Modify the logic to only allow the accounts to be closed when the offer is fully taken:

```
if offer.remaining_amount == 0 {
    // Allow closure
    token::close_account(ctx.accounts.into_offer_close_context())?;
}
```

This approach preserves the data and ensures proper handling of partial fulfillments, protecting the remaining funds and maintaining the integrity of the offer process.

[C-02] ATA initialization causes permanent DoS

Severity

Impact: High

Likelihood: High

Description

In the `create_offer` function, the code uses `init` to initialize an Associated Token Account (ATA) for the Offer PDA to act as a vault for the input token amount:

```
#[account(
    init,
    payer = maker,
    associated_token::mint = input_token_mint,
    associated_token::authority = offer,
    associated_token::token_program = token_program
)]
pub vault_token_account: InterfaceAccount<'info, TokenAccount>,
```

However, an ATA can be created by anyone outside this program using tools like the CLI. Since the `init` constraint requires the ATA to be uninitialized, if the ATA is preemptively created outside the program, this instruction will always fail. This leads to a permanent Denial-of-Service (DoS) condition, preventing the maker from trading tokens.

Additionally, this can result in a loss of protocol fees, as the protocol cannot proceed with the offer creation.

Recommendations

- Use the `init_if_needed` constraint instead of `init`.
- Implement and create idempotent instruction to handle pre-existing ATAs gracefully. This ensures that the program can proceed even if the ATA was created outside the program.

[C-03] `TokenAccounts#fee_wallet` lacks constraints

Severity

Impact: High

Likelihood: High

Description

`take_offer#TokenAccounts#fee_wallet` lacks constraints,taker can pay fee to any account.

```
#[account(
        mut,
        associated_token::mint = input_token_mint,
@>        associated_token::authority = fee_wallet,
        associated_token::token_program = token_program,
    )]
    pub taker_fee_token_account: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(
        mut,
        associated_token::mint = output_token_mint,
$>        associated_token::authority = fee_wallet,
        associated_token::token_program = token_program,
    )]
    pub fee_token_account: Box<InterfaceAccount<'info, TokenAccount>>,


    /// CHECK: Fee wallet validated through fee_token_account constraints
@>    pub fee_wallet: AccountInfo<'info>,
```

link

Recommendations

```
+     #[account(address = fee_config.fee_address)]
      pub fee_wallet: AccountInfo<'info>,
```

8.2. High Findings

[H-01] Missing fee calculation on input token in `take_offer`

Severity

Impact: High

Likelihood: Medium

Description

According to the documentation:

> "A flat fee of 1% is charged to each side of their selected tokens and sent to a treasury wallet controlled by the protocol."

However, in the `take_offer` function, the fee is only charged on the **output token amount**, while the **input token amount** is transferred without any fee deduction, as shown here:

```
token_interface::transfer_checked(
    CpiContext::new_with_signer(
        ctx.accounts.token_program.to_account_info(),
        token_interface::TransferChecked {
            from: ctx.accounts.vault_token_account.to_account_info(),
            mint: ctx.accounts.input_token_mint.to_account_info(),
            to: ctx.accounts.taker_receive_token_account.to_account_info(),
            authority: ctx.accounts.offer.to_account_info(),
        },
        &[seeds],
    ),
    token_amount,
    ctx.accounts.input_token_mint.decimals,
)?;
```

The fee on the output token is calculated and transferred as expected:

```
let fee_amount = expected_payment
    .checked_mul(offer_fee_percentage)
    .unwrap()
    .checked_div(10000)
    .unwrap();

token_interface::transfer_checked(
    CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        token_interface::TransferChecked {
            from: ctx.accounts.taker_payment_token_account.to_account_info(),
            mint: ctx.accounts.output_token_mint.to_account_info(),
            to: ctx.accounts.fee_token_account.to_account_info(),
            authority: ctx.accounts.taker.to_account_info(),
        },
    ),
    fee_amount,
    ctx.accounts.output_token_mint.decimals,
)?;
```

By not charging the fee on the **input token**, the protocol is losing a significant portion of its revenue, deviating from the intended fee model.

Impact

- **loss of funds**: The protocol misses out on half of the intended fees (the input side).

Recommendations

1. **Charge the fee on the input token amount** as well. Modify the code to calculate and transfer the input token fee:

```
let input_fee_amount = token_amount
    .checked_mul(offer_fee_percentage)
    .unwrap()
    .checked_div(10000)
    .unwrap();

let input_after_fee = token_amount.checked_sub(input_fee_amount).unwrap();

// Transfer protocol fee for input token
token_interface::transfer_checked(
    CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        token_interface::TransferChecked {
            from: ctx.accounts.vault_token_account.to_account_info(),
            mint: ctx.accounts.input_token_mint.to_account_info(),
            to: ctx.accounts.fee_token_account.to_account_info(),
            authority: ctx.accounts.offer.to_account_info(),
        },
    ),
    input_fee_amount,
    ctx.accounts.input_token_mint.decimals,
)?;
```

2. Ensure both input and output token fees are properly transferred to the treasury wallet.

This will align the implementation with the protocol's fee structure, ensuring no revenue loss and maintaining user trust.

[H-02] Lack of mint validation in `create_offer` allows arbitrary tokens

Severity

Impact: Medium

Likelihood: High

Description

The `create_offer` function does not enforce any constraints on the `input_token_mint` and `output_token_mint` parameters:

```
/// Input token mint (token being offered)
pub input_token_mint: InterfaceAccount<'info, Mint>,

/// Output token mint (token being requested)
pub output_token_mint: InterfaceAccount<'info, Mint>,

pub fn create_offer(
    ctx: Context<CreateOffer>,
    token_amount: u64,
    expected_amount: u64,
    deadline: i64,
) -> Result<()> {
    // Validate all inputs
    let current_time = Clock::get()?.unix_timestamp;
    require!(deadline > current_time, SwapError::InvalidDeadline);
    require!(token_amount > 0, SwapError::InvalidAmount);

    // Initialize offer parameters
    let offer = &mut ctx.accounts.offer;
    offer.maker = ctx.accounts.maker.key();
    offer.input_token_mint = ctx.accounts.input_token_mint.key();
    offer.output_token_mint = ctx.accounts.output_token_mint.key();
}
```

Although the function initializes the mints and transfers tokens, it does not validate these mints against the `mint_whitelist` defined in the `initialize_admin` function:

```
#[account(
    init,
    payer = admin,
    space = 8 + 4 + (32 * 50),
    seeds = [b"mint_whitelist"],
    bump
)]
pub mint_whitelist: Account<'info, MintWhitelist>,
```

According to the documentation, tradable tokens must be whitelisted by the admin through an admin dashboard. Failure to validate mints allows arbitrary tokens, including those with transfer fees (e.g., tokens implementing the `transfer_fee` extension), to be accepted. These tokens can disrupt the protocol logic, as the amount received after transfer may be less than specified, leading to inconsistencies and loss of funds.

Impact

1. Arbitrary tokens can bypass the whitelist, leading to potential abuse.
2. Tokens with transfer fees can break protocol logic and result in incorrect token amounts.

Recommendations

- Validate both `input_token_mint` and `output_token_mint` against the `mint_whitelist` before proceeding with the offer creation.
- Reject any token mints that are not explicitly listed in the whitelist.

[H-03] The creator of the offer can attack the taker by front-running

Severity

Impact: High

Likelihood: Medium

Description

The `offer` cannot be modified directly after it is created, but it can be modified by re-creating the `offer` after it is closed.

The address of the `offer` is determined by the address of the creator, so the address of the `offer` will not change after the re-creation.

```
#[account(
        init,
        payer = maker,
        space = 8 + 32 + 32 + 32 + 8 + 32 + 8 + 8 + 32 + 1,
@>      seeds = [b"offer", maker.key().as_ref()],
        bump
    )]
    pub offer: Account<'info, Offer>,
```

Therefore, the creator can modify the offer data (such as the price) at any time:

1. Suppose that when a taker takes an offer, the offer price is x(offer_expected_amount/offer_amount),
2. The creator monitors the data on the chain and finds that the offer is about to be traded, immediately modify the order and change the price to 10*x.
3. The creator makes the transaction of modifying the order execution in advance by front-running.
4. The taker took the order at 10 times the expected price.

Recommendations

Add a global index to the seeds of the Offer, or add a price protection parameter when you take the offer.

[H-04] Permanent DOS and loss of funds due to unhandled token dust

Severity

Impact: High

Likelihood: Medium

Description

The `cancel_offer` and `take_offer` functions close the vault token account used to hold the input token amount. However, these functions do not account for any additional tokens transferred to the vault by an attacker. Since non-native token accounts must have a zero balance before being closed, even a minimal token transfer (e.g., 0.0000001 tokens) to the vault would result in the following condition from the `close` function in the Token-2022 program being triggered:

```
if !source_account.base.is_native() && u64::from
  (source_account.base.amount) != 0 {
    return Err(TokenError::NonNativeHasBalance.into());
}
```

This would cause both the `cancel_offer` and `take_offer` functions to revert, leading to:

1. Permanent Denial-of-Service (DoS) for these functions.
2. Permanent lock of funds for the users involved in the offer.

An attacker can execute this exploit at negligible cost, sending a minimal amount of tokens to the vault, which prevents its closure.

Impact

- **Critical Impact:** Permanent DoS for both `cancel_offer` and `take_offer` functions.
- Permanent lock of user funds in the vault.
- Minimal cost for the attacker to execute the exploit.

Recommendations

- Before closing the vault token account, ensure that all the balance is transferred:
  - To the **maker** in the `cancel_offer` function.
  - To the **taker** in the `take_offer` function.

Example adjustment:

```
// Transfer remaining balance before closing the vault
let balance = vault_token_account.amount;

    token_interface::transfer_checked(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            token_interface::TransferChecked {
                from: ctx.accounts.vault_token_account.to_account_info(),
                mint: ctx.accounts.input_token_mint.to_account_info(),
                to: ctx.accounts.maker_token_account.to_account_info
                //(), // in case of cancellation
                authority: offer_auth_info.clone(),
            },
            &[seeds]
        ),
        balance,
        input_token_decimals,
    )?;

// Proceed with closing the account
    token_interface::close_account(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            token_interface::CloseAccount {
                account: ctx.accounts.vault_token_account.to_account_info(),
                destination: ctx.accounts.maker.to_account_info(),
                authority: offer_auth_info,
            },
            &[seeds]
        )
    )?;
```

This ensures that any dust tokens are appropriately transferred, preventing the DoS vulnerability and allowing account closure.

8.3. Medium Findings

[M-01] Whitelisted taker can DoS attack other whitelisted takers

Severity

Impact: Medium

Likelihood: Medium

Description

When `taking _offer`, the `Whitelist` account will be closed:

```
#[account(
        mut,
        seeds = [b"whitelist", offer.maker.as_ref()],
        bump,
        constraint = whitelist.takers.contains(&taker.key
          ()) @ SwapError::TakerNotWhitelisted,
@>      close = maker
    )]
    pub whitelist: Account<'info, Whitelist>,
```

In addition, the order can be partially traded. As a result, the malicious taker could block other takers from trading, and the attacker would only need to close a small number of orders to be unable to continue trading.

Recommendations

Remove the taker from the whitelist after `take_offer` instead of closing the Whitelist account.

[M-02] Same token swap protection missing

Severity

Impact: Medium

Likelihood: Medium

Description

The current implementation allows the creation of offers where input and output tokens are identical. Users could perform wash trading by swapping tokens with themselves, artificially inflating trading volumes. Additionally, this opens vectors for price manipulation since users could create fake trades to influence price oracles or market statistics.

```rust
pub fn create_offer(
    ctx: Context<CreateOffer>,
    token_amount: u64,
    expected_amount: u64,
    deadline: i64,
) -> Result<()> {
    // Validate all inputs
    let current_time = Clock::get()?.unix_timestamp;
    require!(deadline > current_time, SwapError::InvalidDeadline);
    require!(token_amount > 0, SwapError::InvalidAmount);

    // Initialize offer parameters
    let offer = &mut ctx.accounts.offer;
    offer.maker = ctx.accounts.maker.key();
    offer.input_token_mint = ctx.accounts.input_token_mint.key();
    offer.output_token_mint = ctx.accounts.output_token_mint.key();
    offer.token_amount = token_amount;
    offer.expected_total_amount = expected_amount;
    offer.deadline = deadline;

    // Copy protocol configuration
    offer.fee_percentage = ctx.accounts.fee_config.fee_percentage;
    offer.fee_wallet = ctx.accounts.fee_config.fee_address;

    // Transfer tokens to vault with amount validation
    token_interface::transfer_checked(
        CpiContext::new(
            ctx.accounts.token_program.to_account_info(),
            token_interface::TransferChecked {
                from: ctx.accounts.maker_token_account.to_account_info(),
                mint: ctx.accounts.input_token_mint.to_account_info(),
                to: ctx.accounts.vault_token_account.to_account_info(),
                authority: ctx.accounts.maker.to_account_info(),
            },
        ),
        token_amount,
        ctx.accounts.input_token_mint.decimals,
    )?;

    // Update protocol statistics
    let admin_config = &mut ctx.accounts.admin_config;
    admin_config.total_offers += 1;
    admin_config.active_offers += 1;

    // Activate the offer
    offer.status = OfferStatus::Ongoing;

    Ok(())
}
```

Recommendation

Add a comprehensive validation layer that checks for the same token addresses before allowing any swap creation. This should be implemented at both the instruction level and within any helper functions that process swap creation.

8.4. Low Findings

[L-01] Frontrunning the initialization of `admin_config`

`admin_config` is an important data in the system and can only be set once.

If initialize has not been invoked when the contract is deployed, an attacker can preemptively invoke initialize through a front-running attack. In this way, the admin address is set to the attacker's address.

```
pub fn initialize(
    ctx: Context<Initialize>,
    fee_percentage: u64,
    fee_wallet: Pubkey,
    require_whitelist: bool,
    initial_mints: Vec<Pubkey>,
) -> Result<()> {
    require!(fee_percentage <= 10000, SwapError::InvalidFeePercentage);
    require!(initial_mints.len() <= 50, SwapError::TooManyMints);

    // Initialize admin configuration
    let admin_config = &mut ctx.accounts.admin_config;
@>    admin_config.admin = ctx.accounts.admin.key();
    admin_config.total_offers = 0;
    admin_config.active_offers = 0;
    admin_config.completed_offers = 0;
    admin_config.cancelled_offers = 0;
    admin_config.expired_offers = 0;
    admin_config.last_expiry_check = Clock::get()?.unix_timestamp;
   .....
}
```

Initialize only allows calls from fixed addresses.

[L-02] Missing fee address validation in update function

The `fee_address_update` function accepts any Pubkey as a new fee address without performing essential validations. This could lead to setting invalid addresses, system program addresses, program addresses.

```
pub fn fee_address_update(
    ctx: Context<UpdateFeeAddress>,
    new_address: Pubkey
) -> Result<()> {
    ctx.accounts.fee_config.fee_address = new_address;  // No validation
    // performed
    Ok(())
}
```

Consider adding comprehensive address validation:

```
pub fn fee_address_update(
    ctx: Context<UpdateFeeAddress>,
    new_address: Pubkey
) -> Result<()> {

    // Prevent system program address
    require!(!new_address.eq
      (&system_program::ID), SwapError::InvalidFeeAddress);

    // Optional: Add check for address being a normal account
    //(not a PDA or program)
    require!(
        !is_program_address(&new_address),
        SwapError::InvalidFeeAddress
    );

    // Optional: Verify address exists and can receive tokens
    require!(
        validate_fee_address(&new_address),
        SwapError::InvalidFeeAddress
    );

    ctx.accounts.fee_config.fee_address = new_address;
    Ok(())
}
```

[L-03] Missing minimum trade size protection

The protocol doesn't enforce minimum trade size requirements, allowing dust trades that could clog the system and increase operational overhead. Small trades could be used to manipulate protocol statistics or create unnecessary computational load without providing meaningful liquidity.

Implement minimum trade size based on token decimals.

[L-04] State inconsistency due to Solana rollback

One function in the protocol is vulnerable to state inconsistencies in the event of a Solana rollback:

1. **Setting Config Parameters**:
   - Global configuration parameters could become outdated during a Solana rollback
   - Protocol could operate with old, invalid settings
   - Potential for system malfunction or vulnerabilities

Recommendation:

1. **Detect Outdated Configurations**

   - Utilize the `LastRestartSlot` sysvar to check configuration validity
   - Automatically pause protocol if the configuration is outdated
   - Require admin intervention before resuming operations

2. **Add last_updated_slot Field**

   - Include tracking field in bonding curve state
   - Monitor configuration update timestamps

3. **Implement Outdated Configuration Check**

```
fn is_config_outdated(global: &Global) -> Result<bool> {
    let last_restart_slot = LastRestartSlot::get()?;
    Ok(global.last_updated_slot <= last_restart_slot.last_restart_slot)
}
```

[L-05] `offer_id` is not utilized in `create_offer`

The `Offer` struct includes an `offer_id` field, which is meant to uniquely identify each offer. However, in the `create_offer` function, this field is not assigned or used. This oversight reduces the usefulness of the `offer_id`, as there is no unique identifier to track or reference offers externally.

Update the `create_offer` function to assign a unique `offer_id` during offer creation. This can be done using the `Pubkey::find_program_address` function to derive a deterministic ID based on inputs such as the maker's address and the current timestamp:

```
let (offer_id, _) = Pubkey::find_program_address(
    &[b"offer", ctx.accounts.maker.key().as_ref(), &current_time.to_le_bytes
      ()],
    ctx.program_id,
);
offer.offer_id = offer_id;
```

Alternatively, allow the maker to provide the `offer_id` as an argument and validate its uniqueness.

[L-06] Missing validation for an expected amount greater than zero

The `create_offer` function does not validate that the `expected_amount` is greater than zero. This could lead to nonsensical or invalid offers.

Add a check to ensure the `expected_amount` is greater than zero:

```
require!(expected_amount > 0, SwapError::InvalidExpectedAmount);
```

[L-07] Missing Update of Offer Statistics in `take_offer`

In the `take_offer` function, when the entire offer is fulfilled, the following logic is executed:

```
if token_amount == offer_amount {
    // Close vault if fully taken
    token_interface::close_account(CpiContext::new_with_signer(
        ctx.accounts.token_program.to_account_info(),
        token_interface::CloseAccount {
            account: ctx.accounts.vault_token_account.to_account_info(),
            destination: ctx.accounts.maker.to_account_info(),
            authority: ctx.accounts.offer.to_account_info(),
        },
        &[seeds],
    ))?;

    // Mark as completed
    ctx.accounts.offer.status = OfferStatus::Completed;
}
```

While the `OfferStatus` is updated to `Completed`, the associated offer statistics (e.g., `completed_offers` and `ongoing_offers`) are not updated. This can result in inconsistent or inaccurate tracking of the protocol’s overall state, affecting analytics, reporting, and functionality that relies on these statistics.

Impact:

- **Inaccurate statistics**: Metrics such as the number of ongoing and completed offers become incorrect.
- **Operational inefficiency**: Protocol analytics and reporting tools may misrepresent the protocol's performance.
- **Potential user confusion**: Users may see outdated or incorrect offer status in the UI.

Recommendations:

1. **Call** `update_offer_status` after marking the offer as completed to update the protocol’s statistics.

```
if token_amount == offer_amount {
    // Close vault if fully taken
    token_interface::close_account(CpiContext::new_with_signer(
        ctx.accounts.token_program.to_account_info(),
        token_interface::CloseAccount {
            account: ctx.accounts.vault_token_account.to_account_info(),
            destination: ctx.accounts.maker.to_account_info(),
            authority: ctx.accounts.offer.to_account_info(),
        },
        &[seeds],
    ))?;

    // Mark as completed
    ctx.accounts.offer.status = OfferStatus::Completed;

    // Update statistics
    update_offer_status(ctx.accounts.statistics, OfferStatus::Completed)?;
} else {
    // Update remaining amount for partial takes
    ctx.accounts.offer.token_amount = offer_amount.checked_sub
      (token_amount).unwrap();
}
```

2. Ensure that the `update_offer_status` function is properly implemented to increment `completed_offers` and decrement `ongoing_offers`.

3. Write tests to confirm the accuracy of offer statistics after each state change.

This will ensure accurate protocol statistics, maintaining data integrity and improving the user experience.

[L-08] `cancel_offer` requires the maker as a signer, preventing post-expiration cancellations

In the `cancel_offer` function, the following condition allows cancellation of an offer if it is either:

1. Expired (can be canceled by anyone).
2. Ongoing (can only be canceled by the maker):

```
require!(
    current_time > offer_deadline || offer_maker == maker_key,
    SwapError::CannotCancelOffer
);
```

However, the `maker` is required as a signer for this function, as indicated here:

```
#[account(mut)]
pub maker: Signer<'info>,
```

Additionally, the `offer` account has a constraint that enforces the `offer.maker` to match the `maker` signer:

```
#[account(
    mut,
    constraint = offer.maker == maker.key(),

         constraint = offer.status == OfferStatus::Ongoing @ SwapError::InvalidOfferSt
    seeds = [b"offer", maker.key().as_ref()],
    bump,
    close = maker
)]
pub offer: Account<'info, Offer>,
```

This design prevents the second condition (cancellation by anyone after expiration) from being satisfied, as the function always requires the maker to sign, even when the offer is expired.

- Remove the requirement that the `maker` must be the signer for post-expiration cancellations.
- Update the `offer` account constraints as follows:

```
#[account(
    mut,
-   constraint = offer.maker == maker.key(),

         constraint = offer.status == OfferStatus::Ongoing @ SwapError::InvalidOfferSt
    seeds = [b"offer", maker.key().as_ref()],
    bump,
    close = maker
)]
pub offer: Account<'info, Offer>,
```

This adjustment will allow anyone to cancel expired offers while maintaining the requirement for the maker to cancel ongoing offers.

[L-09] Division overflow may result in trading at the price of quote 0

The number of tokens to be paid is calculated by the following the formula:

```
let expected_payment = (token_amount as u128)
        .checked_mul(offer_expected_amount as u128)
        .unwrap()
        .checked_div(offer_amount as u128)
        .unwrap() as u64;
```

expected_payment = token_amount * offer_expected_amount / offer_amount

But the problem is, if token_amount * offer_expected_amount > offer_amount, `expected_payment` will be equal to 0, However, the order is still fulfilled, and taker does not have to pay any token.

`offer_expected_amount` and `offer_amount` are set by the creator when the order is created.

This number is related to the number of decimal places `in_token` and `out_token`, When the decimal values of the two tokens are very different, the attacker(taker) sets a smaller `token_amount` to expected_payment = 0 when `take_offer`.

Test division overflow using https://beta.solpg.io/ Test code:

```rust
use anchor_lang::prelude::*;

// This is your program's public key and it will update
// automatically when you build the project.
declare_id!("31g922MxGWvvitZDQUsaLVrpYF4eoKnLenr3fruNKoXw");

#[program]
mod hello_anchor {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>, data: u64) -> Result<()> {
        let token_amount = 100;
        let offer_expected_amount = 10;
        let offer_amount = 2000;

        let expected_payment = (token_amount as u128)
            .checked_mul(offer_expected_amount as u128)
            .unwrap()
            .checked_div(offer_amount as u128)
            .unwrap() as u64;

        ctx.accounts.new_account.data = expected_payment ;
        msg!
        //("Changed data to: {}!", data); // Message will show up in the tx logs
        Ok(())
    }
}

#[derive(Accounts)]
pub struct Initialize<'info> {
    // We must specify the space in order to initialize an account.
    // First 8 bytes are default account discriminator,
    // next 8 bytes come from NewAccount.data being type u64.
    // (u64 = 64 bits unsigned integer = 8 bytes)
    #[account(init, payer = signer, space = 8 + 8)]
    pub new_account: Account<'info, NewAccount>,
    #[account(mut)]
    pub signer: Signer<'info>,
    pub system_program: Program<'info, System>,
}

#[account]
pub struct NewAccount {
    data: u64,
}
```

```
describe("Test", () => {
  it("initialize", async () => {
    // Generate keypair for the new account
    const newAccountKp = new web3.Keypair();

    // Send transaction
    const data = new BN(42);
    const txHash = await pg.program.methods
      .initialize(data)
      .accounts({
        newAccount: newAccountKp.publicKey,
        signer: pg.wallet.publicKey,
        systemProgram: web3.SystemProgram.programId,
      })
      .signers([newAccountKp])
      .rpc();
    console.log(`Use 'solana confirm -v ${txHash}' to see the logs`);

    // Confirm transaction
    await pg.connection.confirmTransaction(txHash);

    // Fetch the created account
    const newAccount = await pg.program.account.newAccount.fetch(
      newAccountKp.publicKey
    );

    console.log("On-chain data is:", newAccount.data.toString());

    // Check whether the data on-chain is equal to local 'data'
    //assert(data.eq(newAccount.data));
  });
});
```

Recommendation:

```
// Calculate proportional payment based on taken amount
    let expected_payment = (token_amount as u128)
        .checked_mul(offer_expected_amount as u128)
        .unwrap()
        .checked_div(offer_amount as u128)
        .unwrap() as u64;

+    require!(expected_payment > 0, SwapError::InsufficientAmount);
```

[L-10] Expiring offers without closing the offer PDA causes DoS

The `update_offer_status` function allows the admin to mark an offer as `Expired` :

```
admin_config.update_offer_status(Some(offer.status), OfferStatus::Expired);

// Mark offer as expired
offer.status = OfferStatus::Expired;
```

This sets the offer state to `Expired`, but both the `cancel_offer` and `take_offer` functions require the offer status to remain `Ongoing`. If the offer is marked as `Expired`, both functions will return errors due to status mismatches:

```
#[account(
    mut,
    constraint = offer.maker == maker.key(),

         constraint = offer.status == OfferStatus::Ongoing @ SwapError::InvalidOfferSt
    seeds = [b"offer", maker.key().as_ref()],
    bump,
    close = maker
)]
pub offer: Account<'info, Offer>,
```

This behavior creates a Denial-of-Service condition where the maker’s funds are locked permanently, and the maker is unable to create new offers , and no taker can take the offer.

Ensure that the offer is properly canceled "closed" after being marked as `Expired` by the admin to prevent the funds from being locked and to allow the maker to create new offers.

[L-11] Unreliable event logging due to Solana log truncation

Solana nodes truncate logs larger than 10 KB by default, making regular events emitted via the `emit!` macro unreliable for large data sets.

Impact

- Potential loss of critical event data
- Inconsistent off-chain data processing
- Reduced transparency and audibility of on-chain actions

Current Implementation

The code currently uses both `emit!` and `emit_cpi!`:

```
emit!(OfferTaken {
        offer_id: offer.offer_id,
        maker: offer.maker,
        taker: ctx.accounts.core.taker.key(),
        input_token_amount,
        payment_amount,
        fee_amount,
        remaining_amount: new_remaining,
        input_token_mint: ctx.accounts.token.input_token_mint.key(),
        output_token_mint: ctx.accounts.token.output_token_mint.key(),
    });
```

Recommended Solution

Remove the `emit!` macro and keep only `emit_cpi!`:

```
emit_cpi!(OfferTaken {
        offer_id: offer.offer_id,
        maker: offer.maker,
        taker: ctx.accounts.core.taker.key(),
        input_token_amount,
        payment_amount,
        fee_amount,
        remaining_amount: new_remaining,
        input_token_mint: ctx.accounts.token.input_token_mint.key(),
        output_token_mint: ctx.accounts.token.output_token_mint.key(),
    });
```