ExVul

# SMART CONTRACT AUDIT REPORT

DeFAI Swap & Stake Smart Contract

AUGUST 2025

## Contents

# 1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **DeFAI** to review smart contract implementation. The assessment was conducted in accordance with our systematic approach to evaluate potential security issues based upon customer requirement. The report provides detailed recommendations to resolve the issue and provide additional suggestions or recommendations for improvement.

The outcome of the assessment outlined in chapter 3 provides the system's owners a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendations that will resolve the underlying technical issue.

## 1.1 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10] which is the gold standard in risk assessment using the following risk models:

- **Likelihood**: represents how likely a particular vulnerability is to be uncovered and exploited in the wild.
- **Impact**: measures the technical loss and business damage of a successful attack.
- **Severity**: determine the overall criticality of the risk.

Likelihood can be: High, Medium and Low and impact are categorized into: High, Medium, Low, Informational. Severity is determined by likelihood and impact and can be classified into five categories accordingly: Critical, High, Medium, Low, Informational shown in table 1.1.

| Likelihood | Informational | Low | Medium | High |
|---|---|---|---|---|
| *High* | INFO | MEDIUM | HIGH | CRITICAL |
| *Medium* | INFO | LOW | MEDIUM | HIGH |
| *Low* | INFO | LOW | LOW | MEDIUM |

IMPACT

**Table 1.1 Overall Risk Severity**

To evaluate the risk, we will be going through a list of items, and each would be labelled with a severity category. The audit was performed with a systematic approach guided by a comprehensive assessment list carefully designed to identify known and impactful security issues. If our tool or analysis does not identify any issue, the contract can be considered safe regarding the assessed item. For any discovered issue, we might further deploy contracts on our private test environment and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.2.

- **Basic Coding Bugs**: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- **Code and business security testing**: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- **Additional Recommendations**: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

| **Category** | **Assessment Item** |
|---|---|
| **Basic Coding Assessment** | • Apply Verification Control<br>• Authorization Access Control<br>• Forged Transfer Vulnerability<br>• Forged Transfer Notification<br>• Numeric Overflow<br>• Transaction Rollback Attack<br>• Transaction Block Stuffing Attack<br>• Soft Fail Attack<br>• Hard Fail Attack<br>• Abnormal Memo<br>• Abnormal Resource Consumption<br>• Secure Random Number |

| | |
|---|---|
| **Advanced Source Code Scrutiny** | • Asset Security<br>• Cryptography Security<br>• Business Logic Review<br>• Source Code Functional Verification<br>• Account Authorization Control<br>• Sensitive Information Disclosure<br>• Circuit Breaker<br>• Blacklist Control<br>• System API Call Analysis<br>• Contract Deployment Consistency Check<br>• Abnormal Resource Consumption |
| **Additional Recommendations** | • Semantic Consistency Checks<br>• Following Other Best Practices |

**Table 1.2: The Full List of Assessment Items**

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development.

# 2. FINDINGS OVERVIEW

## 2.1 Project Info And Contract Address

| Project Name | Audit Time | Language |
|---|---|---|
| DeFAI Swap & Stake | 21/07/2025 - 13/08/2025 | Rust |

| Repository |
|---|
| https://github.com/defaiza/audit.git |

| Commit Hash |
|---|
| bbf88147743821d60bdbcf335e25e8ac9159f441 |

## 2.2 Summary

| Severity | Found |
|---|---|
| CRITICAL | 0 |
| HIGH | 6 |
| MEDIUM | 6 |
| LOW | 3 |
| INFO | 1 |

### 2.3 Key Findings

| Severity | Findings Title | Status |
|---|---|---|
| HIGH | Unvalidated Accounts in fund_escrow Lead to State Pollution and Protocol Insolvency | Fixed |
| HIGH | Unvalidated Accounts in unstake_tokens Allow Penalty Interception | Fixed |
| HIGH | Unvalidated Escrow Account in compound_rewards Leads to State Pollution | Fixed |
| HIGH | Missing Treasury Account Validation Allows Tax Funds to be Redirected | Fixed |
| HIGH | Missing NFT Mint Validation Allows Unauthorized Vesting Claims | Fixed |
| HIGH | Token-2022 Funds Permanently Locked Due to AdminWithdraw Token Standard Mismatch | Fixed |
| MEDIUM | Combination of State Update Leads to Permanent Loss of Funds | Fixed |
| MEDIUM | Penalty Logic in unstake_tokens Can Be Bypassed | Fixed |
| MEDIUM | OG Tier 0 and Paid Tier 0 Share Supply Pool | Fixed |
| MEDIUM | Tax Reset Time Comparison Inconsistency | Fixed |
| MEDIUM | Pause Mechanism Not Enforced in Critical Functions | Fixed |
| MEDIUM | VRF Switch and Results Not Being Used | Fixed |
| LOW | Initialization Functions Lack Authority Validation | Fixed |
| LOW | Missing Admin Authorization Validation in Whitelist Initialization | Fixed |
| LOW | OLD DEFAI Swap Breaks Tax Reset Mechanism | Fixed |
| INFO | Enable VRF Function Lacks State Validation | Fixed |

Table 2.3: Key Audit Findings

## 3. DETAILED DESCRIPTION OF FINDINGS

### 3.1 Unvalidated Accounts in fund_escrow Lead to State Pollution and Protocol Insolvency

**SEVERITY:** HIGH **STATUS:** Fixed

### PATH:

`defai_staking/src/lib.rs::fund_escrow()`

### DESCRIPTION:

The fund_escrow function, which allows anyone to add funds to the reward pool, fails to apply strict constraints to the escrow_token_account and defai_mint accounts passed into its context, FundEscrow.

An attacker can exploit this by passing in a malicious token (evil_mint) they created, along with its corresponding token account (ATA). Although the genuine reward_escrow PDA cannot be forged, the program will erroneously add the amount of the malicious tokens to the total_balance state of the genuine reward_escrow account. This leads to a pollution of the protocol's state.

```
#[derive(Accounts)]
pub struct FundEscrow<'info> {
    #[account(mut)]
    pub reward_escrow: Account<'info, RewardEscrow>,

    #[account(mut)]
    pub escrow_token_account: InterfaceAccount<'info, TokenAccount>, // Attacker controlled

    #[account(mut)]
    pub funder_token_account: InterfaceAccount<'info, TokenAccount>, // Attacker controlled

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

    pub defai_mint: InterfaceAccount<'info, Mint>, // Attacker controlled
    pub token_program: Interface<'info, TokenInterface>,
}
```

## IMPACT:

This is a critical vulnerability that allows an attacker to destroy the protocol's economic model through state pollution, ultimately enabling the theft of all users' stake funds. The maliciously inflated reward_escrow.total_balance invalidates all functions that rely on this value for security checks.

## RECOMMENDATIONS:

Add strict account constraints to the FundEscrow context to ensure that all incoming accounts match the authoritative addresses stored in the program's state:

```
#[derive(Accounts)]
pub struct FundEscrow<'info> {
    pub program_state: Account<'info, ProgramState>,

    #[account(
        mut,
        seeds = [b”reward-escrow”, program_state.key().as_ref()],
        bump
    )]
    pub reward_escrow: Account<'info, RewardEscrow>,

    #[account(
        mut,
        seeds = [b”escrow-vault”, program_state.key().as_ref()],
        bump,
        token::authority = reward_escrow,
        token::mint = defai_mint,
    )]
    pub escrow_token_account: InterfaceAccount<'info, TokenAccount>,

    #[account(
        constraint = defai_mint.key() == program_state.defai_mint @
            StakingError::InvalidMint
    )]
    pub defai_mint: InterfaceAccount<'info, Mint>,
}
```

### 3.2 Unvalidated Accounts in unstake_tokens Allow Penalty Interception

**SEVERITY:** HIGH **STATUS:** Fixed

**PATH:**

`defai_staking/src/lib.rs::unstake_tokens()`

**DESCRIPTION:**

The unstake_tokens function, when processing penalties generated from early withdrawals, fails to validate whether the receiving account for the penalty (escrow_token_account) is the official reward pool address. The function trusts the account address provided by the user, which allows any user performing an early unstake to supply an account they control, thereby intercepting the penalty fee that should have gone to the protocol.

```
#[derive(Accounts)]
pub struct UnstakeTokens<'info> {
    #[account(mut)]
    pub reward_escrow: Account<'info, RewardEscrow>, // Not validated as the official PDA

    #[account(mut)]
    pub escrow_token_account: InterfaceAccount<'info, TokenAccount>, // Attacker-controlled
}

// Penalty transfer logic
if penalty > 0 {
    let transfer_penalty_ctx = CpiContext::new_with_signer(
        TransferChecked {
            from: ctx.accounts.stake_vault.to_account_info(),
            to: ctx.accounts.escrow_token_account.to_account_info(), // Destination: Attacker's account
            authority: ctx.accounts.stake_vault.to_account_info(),
            mint: ctx.accounts.defai_mint.to_account_info(),
        },
        signer,
    );
    transfer_checked(transfer_penalty_ctx, penalty,
        ctx.accounts.defai_mint.decimals)?;
}
```

## IMPACT:

This vulnerability allows users to directly steal penalty fees that are intended as protocol revenue. Any user executing an early unstake can exploit this method to intercept their own penalty payment, breaking the economic model designed to discourage short-term speculation.

## RECOMMENDATIONS:

Add strict account constraints to the UnstakeTokens context to ensure that the provided reward_escrow and escrow_token_account are the official, unique PDA accounts of the protocol.

### 3.3 Unvalidated Escrow Account in compound_rewards Leads to State Pollution

**SEVERITY:** HIGH **STATUS:** Fixed

## PATH:

`defai_staking/src/lib.rs::compound_rewards()`

## DESCRIPTION:

The compound_rewards function has a flaw where it fails to validate the address of the incoming reward_escrow account. It only checks if the account is mutable but does not use a PDA constraint to verify that it is the official, program-controlled escrow account.

This allows a caller to pass in a self-created, fake reward_escrow account containing false data to execute the compound instruction.

```
#[derive(Accounts)]
pub struct CompoundRewards<'info> {
    #[account(mut)] // Address of reward_escrow is not validated here
    pub reward_escrow: Account<'info, RewardEscrow>,
}

// The check passes because it uses the balance from the fake account
require!(
    ctx.accounts.reward_escrow.total_balance >= total_unclaimed,
    StakingError::InsufficientEscrowBalance
);

// The attacker's stake and the global stake are incorrectly increased
user_stake.staked_amount = user_stake.staked_amount
    .checked_add(total_unclaimed).unwrap();

program_state.total_staked = program_state.total_staked
    .checked_add(total_unclaimed).unwrap();
```

## IMPACT:

This vulnerability pollutes the protocol's core state by allowing attackers to use fake escrow accounts with inflated balances, causing state inconsistencies between user balances and actual protocol funds.

## RECOMMENDATIONS:

Add a strict PDA constraint to the reward_escrow account within the CompoundRewards context:

```
#[derive(Accounts)]
pub struct CompoundRewards<'info> {
    #[account(mut)]
    pub program_state: Account<'info, ProgramState>,

    #[account(
        mut,
        seeds = [b”reward-escrow”, program_state.key().as_ref()],
        bump = program_state.escrow_bump
    )]
    pub reward_escrow: Account<'info, RewardEscrow>,
}
```

### 3.4 Missing Treasury Account Validation Allows Tax Funds to be Redirected

**SEVERITY:** HIGH **STATUS:** Fixed

**PATH:**

defai_swap/src/lib.rs::swap_defai_for_pnft_v6()

**DESCRIPTION:**

The swap_defai_for_pnft_v6 function accepts user-provided treasury and escrow accounts without validation, allowing malicious users to redirect tax funds to arbitrary accounts they control.

```
#[derive(Accounts)]
pub struct SwapDefaiForPnftV6<'info> {
    #[account(mut)]
    pub treasury_defai_ata: Box<InterfaceAccount<'info,
       TokenAccount2022>>,
    #[account(mut)]
    pub escrow_defai_ata: Box<InterfaceAccount<'info, TokenAccount2022>>,
}

pub fn swap_defai_for_pnft_v6(ctx: Context<SwapDefaiForPnftV6>, tier: u8,
    ...) -> Result<()> {
    let tax_amount = (price as u128)
        .checked_mul(user_tax.tax_rate_bps as u128)
        .ok_or(ErrorCode::MathOverflow)?
        .checked_div(10000)
        .ok_or(ErrorCode::MathOverflow)? as u64;

    let cpi_ctx_tax = CpiContext::new(
        ctx.accounts.token_program_2022.to_account_info(),
        TransferChecked {
            from: ctx.accounts.user_defai_ata.to_account_info(),
            to: ctx.accounts.treasury_defai_ata.to_account_info(),  // No
               validation
            authority: ctx.accounts.user.to_account_info(),
            mint: ctx.accounts.defai_mint.to_account_info(),
        },
    );
    token22::transfer_checked(cpi_ctx_tax, tax_amount, 6)?;
}
```

## IMPACT:

Tax funds can be redirected to attacker-controlled accounts instead of the legitimate treasury, causing the protocol to lose revenue while attackers gain additional funds.

## RECOMMENDATIONS:

Add validation to ensure treasury and escrow accounts match the configured addresses in the CollectionConfig.

### 3.5 Missing NFT Mint Validation Allows Unauthorized Vesting Claims

**SEVERITY:** HIGH

**STATUS:** Fixed

## PATH:

`defai_swap/src/lib.rs::claim_vested_v6(), reroll_bonus_v6()`

## DESCRIPTION:

The claim_vested_v6 and reroll_bonus_v6 functions only verify NFT ownership and amount, but fail to validate that the provided NFT ATA corresponds to the correct NFT mint, allowing attackers to use any NFT to claim vesting rewards for a different NFT.

```rust
pub fn claim_vested_v6(ctx: Context<ClaimVestedV6>) -> Result<()> {
    require!(
        ctx.accounts.user_nft_ata.owner == ctx.accounts.user.key() &&
        ctx.accounts.user_nft_ata.amount == 1,
        ErrorCode::NoNft
    );

    // Process vesting without verifying NFT mint
    let vesting_state = &mut ctx.accounts.vesting_state;
}

pub fn reroll_bonus_v6(ctx: Context<RerollBonusV6>) -> Result<()> {
    // Same vulnerability
    require!(
        ctx.accounts.user_nft_ata.owner == ctx.accounts.user.key() &&
        ctx.accounts.user_nft_ata.amount == 1,
        ErrorCode::NoNft
    );
}
```

## IMPACT:

Attackers can use any NFT they own to claim vesting rewards for a different NFT, completely bypassing NFT ownership verification for vesting and reroll functions.

## RECOMMENDATIONS:

Add mint validation to ensure the NFT ATA corresponds to the correct NFT:

```
pub fn claim_vested_v6(ctx: Context<ClaimVestedV6>) -> Result<()> {
    require!(
        ctx.accounts.user_nft_ata.owner == ctx.accounts.user.key() &&
        ctx.accounts.user_nft_ata.amount == 1 &&
        ctx.accounts.user_nft_ata.mint == ctx.accounts.nft_mint.key(),
        ErrorCode::NoNft
    );
}
```

### 3.6 Token-2022 Funds Permanently Locked Due to AdminWithdraw Token Standard Mismatch

**SEVERITY:** HIGH **STATUS:** Fixed

**PATH:**

`defai_swap/src/lib.rs::admin_withdraw()`

**DESCRIPTION:**

The admin_withdraw function only supports standard SPL Token program and cannot operate Token-2022 accounts, causing DEFAI fee funds deposited via swap_defai_for_pnft_v6 to be permanently locked.

```
// SwapDefaiForPnftV6 uses Token-2022
#[derive(Accounts)]
pub struct SwapDefaiForPnftV6<'info> {
    pub token_program_2022: Program<'info, Token2022>,  // Token-2022
    #[account(mut)]
    pub escrow_defai_ata: Box<InterfaceAccount<'info, TokenAccount2022>>,
}

// But AdminWithdraw only supports standard SPL Token
#[derive(Accounts)]
pub struct AdminWithdraw<'info> {
    #[account(mut)]
    pub source_vault: Account<'info, TokenAccount>,
    pub token_program: Program<'info, Token>,        // Cannot operate Token-2022
}

pub fn admin_withdraw(ctx: Context<AdminWithdraw>, amount: u64) -> Result<()> {
    let cpi_ctx = CpiContext::new_with_signer(
        ctx.accounts.token_program.to_account_info(),  // Standard SPL Token
        Transfer {
            from: ctx.accounts.source_vault.to_account_info(),
            to: ctx.accounts.dest.to_account_info(),
            authority: ctx.accounts.escrow.to_account_info(),
        },
        &[&escrow_seeds[..]],
    );
    token::transfer(cpi_ctx, amount)?;
}
```

## IMPACT:

All DEFAI fees from swap_defai_for_pnft_v6 are permanently locked and cannot be withdrawn by admin due to token standard mismatch.

## RECOMMENDATIONS:

Add Token-2022 withdrawal function or modify existing admin_withdraw to handle both token standards by adding an is_token2022 parameter.

### 3.7 Combination of State Update Leads to Permanent Loss of Funds

**SEVERITY:** MEDIUM **STATUS:** Fixed

## PATH:

`defai_staking/src/lib.rs::stake_tokens(), update_defai_mint()`

## DESCRIPTION:

The protocol has a critical design flaw: the core stake_tokens function lacks address validation for the Vault account, while the update_defai_mint function allows an administrator to change the official token recorded in the program's state. This combination creates a scenario where funds can be permanently locked.

```
#[derive(Accounts)]
pub struct StakeTokens<'info> {
    #[account(mut)] // Missing PDA validation allows any vault address
    pub stake_vault: InterfaceAccount<'info, TokenAccount>,
}

// Withdrawal fails because signing logic is hardcoded
let program_state_key = ctx.accounts.program_state.key(); // This key is
    constant
let seeds = &[
    b”stake-vault”,
    program_state_key.as_ref(), // This part of the seed is constant
    &[ctx.accounts.program_state.vault_bump],
];
let signer = &[&seeds[..]]; // The signer always represents the original
    vault
```

During a routine operation intended to upgrade the protocol to support a new token:

1. **Staking Succeeds**: A project owner can create a new vault associated with a new token and guide users to stake into it via the frontend. Because the stake_tokens function does not validate the vault address, this transaction completes successfully.
2. **Withdrawal Fails**: When a user attempts to withdraw their assets, the unstake_tokens function will fail because the program's withdrawal signing logic is hardcoded to generate a signature

only for the original, official vault.

## IMPACT:

All funds deposited into the new vault will be permanently locked and unrecoverable.

## RECOMMENDATIONS:

1. **Enforce Account Validation**: Add a PDA constraint to strictly validate the vault address in all functions that interact with funds.
2. **Remove Risky Function**: Completely remove the update_defai_mint function. A token migration should be handled through a comprehensive solution that includes a fund migration strategy.

### 3.8 Penalty Logic in unstake_tokens Can Be Bypassed

**SEVERITY:** MEDIUM **STATUS:** Fixed

**PATH:**

`defai_staking/src/lib.rs::calculate_unstake_penalty(), stake_tokens()`

**DESCRIPTION:**

The calculate_unstake_penalty function determines the penalty based on the stake_timestamp (the time of the initial stake). If a user adds more funds via stake_tokens after their initial stake, the new funds are considered as "old" as the first deposit because the stake_timestamp is only set once and not updated on subsequent stakes.

```rust
fn calculate_unstake_penalty(
    stake_timestamp: i64,
    current_timestamp: i64,
    amount: u64,
) -> Result<u64> {
    let days_staked = (current_timestamp - stake_timestamp) / 86400;

    let penalty_bps = if days_staked < 30 {
        200  // 2%
    } else if days_staked < 90 {
        100  // 1%
    } else {
        0    // No penalty
    };

    Ok((amount as u128 * penalty_bps as u128 / BASIS_POINTS as u128) as
        u64)
}

pub fn stake_tokens(/* ... */) -> Result<()> {
    if user_stake.owner == Pubkey::default() {
        user_stake.stake_timestamp = clock.unix_timestamp;
    } else {
        // Existing user adds to stake - timestamp NOT updated
        user_stake.staked_amount =
            user_stake.staked_amount.checked_add(amount).unwrap();
    }
    Ok(())
}
```

## IMPACT:

This vulnerability allows an attacker to bypass the early unstake penalty by first staking a small amount to “age” their account. After waiting 90 days to gain penalty-free status, they can then deposit a much larger sum, earn short-term rewards on this large capital, and immediately withdraw everything with zero penalty.

## RECOMMENDATIONS:

Add a last_stake_timestamp field to the UserStake struct. This timestamp should be updated every time a user adds to their stake, and the penalty calculation should use this most recent timestamp.

### 3.9 OG Tier 0 and Paid Tier 0 Share Supply Pool

**SEVERITY:** MEDIUM **STATUS:** Fixed

## PATH:

defai_swap/src/lib.rs::swap_og_tier0_for_pnft_v6(), swap_defai_for_pnft_v6()

## DESCRIPTION:

The swap_og_tier0_for_pnft_v6 and swap_defai_for_pnft_v6 functions both use the same tier_minted[0] counter and tier_supplies[0] limit, allowing paid users to exhaust the supply before OG Tier 0 holders can claim their reserved allocation.

```rust
pub fn swap_og_tier0_for_pnft_v6(ctx: Context<SwapOgTier0ForPnftV6>, ...)
    -> Result<()> {
    require!(
        config.tier_minted[0] < config.tier_supplies[0],
        ErrorCode::NoLiquidity
    );

    config.tier_minted[0] += 1;
}

pub fn swap_defai_for_pnft_v6(ctx: Context<SwapDefaiForPnftV6>, tier: u8,
    ...) -> Result<()> {
    require!(tier < 5, ErrorCode::InvalidTier);

    require!(
        config.tier_minted[tier as usize] < config.tier_supplies[tier as
            usize],
        ErrorCode::NoLiquidity
    );

    config.tier_minted[tier as usize] += 1;
}
```

## IMPACT:

Paid users can exhaust the Tier 0 supply before OG Tier 0 holders claim their reserved allocation, preventing OG Tier 0 holders from claiming their free NFTs and creating unfair distribution where paid users have priority over reserved OG allocations.

## RECOMMENDATIONS:

Separate the supply pools for OG Tier 0 and paid Tier 0 users by adding og_tier_0_supply and og_tier_0_minted fields to CollectionConfig, and modify the logic to check remaining supply after reserving for OG holders.

### 3.10 Tax Reset Time Comparison Inconsistency

**SEVERITY:** MEDIUM **STATUS:** Fixed

## PATH:

`defai_swap/src/lib.rs::reset_user_tax(), swap_defai_for_pnft_v6()`

## DESCRIPTION:

The reset_user_tax and swap_defai_for_pnft_v6 functions use inconsistent time comparison logic for tax reset, allowing users who call reset_user_tax first to pay significantly lower taxes than users who directly call swap_defai_for_pnft_v6 at the same timestamp.

```rust
pub fn reset_user_tax(ctx: Context<ResetUserTax>) -> Result<()> {
    let user_tax_state = &mut ctx.accounts.user_tax_state;
    let now = Clock::get()?.unix_timestamp;

    require!(
        now >= user_tax_state.last_swap_timestamp + TAX_RESET_DURATION,
            // Uses >=
        ErrorCode::TaxResetTooEarly
    );

    user_tax_state.tax_rate_bps = INITIAL_TAX_BPS;  // Reset to 5%
    user_tax_state.swap_count = 0;
}

pub fn swap_defai_for_pnft_v6(ctx: Context<SwapDefaiForPnftV6>, tier: u8,
   ...) -> Result<()> {
    // Check and reset tax if 24 hours passed
    if clock.unix_timestamp - user_tax.last_swap_timestamp >
        TAX_RESET_DURATION {  // Uses >
        user_tax.tax_rate_bps = INITIAL_TAX_BPS;
        user_tax.swap_count = 0;
    }
}
```

## IMPACT:

At exactly 24 hours after the last swap, users who call reset_user_tax first pay 5% tax, while users who directly call swap_defai_for_pnft_v6 at the same timestamp pay up to 30% tax, creating an unfair 25% tax difference for identical timing conditions.

## RECOMMENDATIONS:

Standardize time comparison logic to use >= in both functions for consistency.

### 3.11 Pause Mechanism Not Enforced in Critical Functions

**SEVERITY:** MEDIUM **STATUS:** Fixed

## PATH:

defai_swap/src/lib.rs::swap_defai_for_pnft_v6(), claim_vested_v6(), redeem_v6()

## DESCRIPTION:

The contract provides pause/unpause functions but fails to check config.paused in core instructions like swap, reroll, claim, and redeem, allowing protocol operations to continue even when paused.

```
pub fn swap_defai_for_pnft_v6(ctx: Context<SwapDefaiForPnftV6>, tier: u8,
   ...) -> Result<()> {
    // Missing pause check - protocol continues even when paused
    let config = &mut ctx.accounts.collection_config;
}

pub fn claim_vested_v6(ctx: Context<ClaimVestedV6>) -> Result<()> {
    // Missing pause check - users can still claim when protocol is paused
    let vesting_state = &mut ctx.accounts.vesting_state;
}

pub fn redeem_v6(ctx: Context<RedeemV6>) -> Result<()> {
    // Missing pause check - redemptions continue even when paused
    let bonus_state = &mut ctx.accounts.bonus_state;
}
```

## IMPACT:

Protocol operations continue even when admin has paused the system, making the emergency pause mechanism ineffective for critical functions and allowing users to continue swapping, claiming, and redeeming during emergency situations.

## RECOMMENDATIONS:

Add pause validation to all critical functions by checking the config.paused flag at the beginning of each function.

### 3.12 VRF Switch and Results Not Being Used

**SEVERITY:** MEDIUM **STATUS:** Fixed

**PATH:**

`defai_swap/src/lib.rs::enable_vrf(), swap_defai_for_pnft_v6()`

**DESCRIPTION:**

The enable_vrf function only sets cfg.vrf_enabled flag, but core functions continue using weak randomness generate_secure_random instead of VRF results. The vrf_state.result_buffer is never read or used in the actual random generation.

```rust
pub fn enable_vrf(ctx: Context<UpdateConfig>) -> Result<()> {
    let cfg = &mut ctx.accounts.config;
    cfg.vrf_enabled = true;  // Only sets flag, no actual VRF usage
}

pub fn swap_defai_for_pnft_v6(ctx: Context<SwapDefaiForPnftV6>, tier: u8,
   ...) -> Result<()> {
    let random_value = generate_secure_random(
        &ctx.accounts.user.key(),
        &ctx.accounts.nft_mint.key(),
        &clock,
        &blockhash_bytes,
    );
}
```

**IMPACT:**

VRF functionality is effectively disabled despite being “enabled”, with core functions continuing to use predictable randomness from recent_blockhashes while VRF infrastructure exists but provides no security benefit.

**RECOMMENDATIONS:**

Enforce VRF usage when enabled and use VRF results:

```
pub fn swap_defai_for_pnft_v6(ctx: Context<SwapDefaiForPnftV6>, tier: u8,
    ...) -> Result<()> {
    if ctx.accounts.config.vrf_enabled {
        require!(vrf_state.result_buffer != [0u8; 32],
            ErrorCode::VrfNotReady);
        let random_value = generate_vrf_random(
            &vrf_state.result_buffer,
            &ctx.accounts.user.key(),
            &ctx.accounts.nft_mint.key(),
        );
    } else {
        // Use weak randomness only when VRF disabled
        let random_value = generate_secure_random(...);
    }
}
```

### 3.13 Initialization Functions Lack Authority Validation

**SEVERITY:** LOW **STATUS:** Fixed

**PATH:**

`defai_staking/src/lib.rs::initialize_program(), initialize_escrow()`

**DESCRIPTION:**

The critical initialization functions initialize_program and initialize_escrow lack validation for the caller's identity. They accept any signer as the authority without verifying if the signer is the intended protocol administrator.

```
#[derive(Accounts)]
pub struct InitializeProgram<'info> {
    #[account(
        init,
        payer = authority, // payer can be anyone
    )]
    pub program_state: Account<'info, ProgramState>,

    #[account(mut)]
    pub authority: Signer<'info>, // Lacks validation for the authority's
       identity
}

#[derive(Accounts)]
pub struct InitializeEscrow<'info> {
    #[account(
        mut,
        seeds = [b”program-state”],
        bump
    )]
    pub program_state: Account<'info, ProgramState>,

    #[account(mut)]
    pub authority: Signer<'info>, // Lacks validation for the authority's
       identity
}
```

## IMPACT:

A malicious or random user could front-run the actual project administrator and call the initialize_program function first, causing the protocol to be initialized with an incorrect admin address and potentially forcing the project team to redeploy the entire contract.

## RECOMMENDATIONS:

Add proper authority validation to these functions to ensure only the legitimate protocol administrator can perform these critical initializations.

### 3.14 Missing Admin Authorization Validation in Whitelist Initialization

**SEVERITY:** LOW **STATUS:** Fixed

## PATH:

`defai_swap/src/lib.rs::initialize_whitelist()`

## DESCRIPTION:

The initialize_whitelist function lacks admin authorization validation. The function accepts any signer as admin without verifying if they are the actual protocol administrator.

```
pub fn initialize_whitelist(ctx: Context<InitializeWhitelist>) ->
    Result<()> {
     let whitelist = &mut ctx.accounts.whitelist;
     whitelist.root = WHITELIST_ROOT;
     whitelist.claimed_count = 0;
     Ok(())
}
```

## IMPACT:

Any user can initialize the whitelist by providing a signature and paying account creation fees.

## RECOMMENDATIONS:

Add config account to InitializeWhitelist context and implement proper authorization validation:

```
pub fn initialize_whitelist(ctx: Context<InitializeWhitelist>) ->
    Result<()> {
     require_keys_eq!(ctx.accounts.admin.key(), ctx.accounts.config.admin,
        ErrorCode::Unauthorized);

     let whitelist = &mut ctx.accounts.whitelist;
     whitelist.root = WHITELIST_ROOT;
     whitelist.claimed_count = 0;
     Ok(())
}
```

### 3.15 OLD DEFAI Swap Breaks Tax Reset Mechanism

**SEVERITY:** LOW **STATUS:** Fixed

## PATH:

`defai_swap/src/lib.rs::swap_old_defai_for_pnft_v6()`

## DESCRIPTION:

The swap_old_defai_for_pnft_v6 function claims “No tax for old DEFAI swaps” but still updates user_tax.last_swap_timestamp, which breaks the 24-hour tax reset mechanism. This creates an unfair situation where users who use OLD DEFAI tokens are penalized with higher tax rates when they later use DEFAI tokens.

```
pub fn swap_old_defai_for_pnft_v6(
    ctx: Context<SwapOldDefaiForPnftV6>,
    tier: u8,
    _metadata_uri: String,
    _name: String,
    _symbol: String,
) -> Result<()> {
    // Update user tax state
    user_tax.swap_count += 1;
    user_tax.last_swap_timestamp = clock.unix_timestamp;  // Breaks reset
        mechanism
}
```

**Scenario:** 1. User performs 25 DEFAI swaps, reaching 30% tax rate 2. Waits 24 hours for tax reset 3. Calls swap_old_defai_for_pnft_v6 (no tax charged) 4. last_swap_timestamp gets updated, breaking the reset window 5. Calls swap_defai_for_pnft_v6 again 6. Tax rate remains at 30% instead of resetting to 5%

## IMPACT:

Users using OLD DEFAI are penalized with higher taxes due to the logic contradiction where “No tax” functions still affect tax state.

## RECOMMENDATIONS:

Add tax reset logic to swap_old_defai_for_pnft_v6 or remove the timestamp update entirely since OLD DEFAI swaps should not affect the tax state.

### 3.16 Enable VRF Function Lacks State Validation

**SEVERITY:** INFO **STATUS:** Fixed

## PATH:

`defai_swap/src/lib.rs::enable_vrf()`

## DESCRIPTION:

The enable_vrf function does not check the current state of vrf_enabled before setting it to true, allowing redundant calls even when VRF is already enabled. This creates inconsistent behavior compared to other state management functions like pause and unpause.

```
pub fn enable_vrf(ctx: Context<UpdateConfig>) -> Result<()> {
    require_keys_eq!(ctx.accounts.admin.key(), ctx.accounts.config.admin,
        ErrorCode::Unauthorized);

    let cfg = &mut ctx.accounts.config;
    cfg.vrf_enabled = true;  // No state validation

    msg!(”VRF enabled for swap program”);

    emit!(AdminAction {
        admin: ctx.accounts.admin.key(),
        action: ”Enable VRF”.to_string(),
        timestamp: Clock::get()?.unix_timestamp,
    });

    Ok(())
}

// Compare with pause function that has proper state validation
pub fn pause(ctx: Context<UpdateConfig>) -> Result<()> {
    require_keys_eq!(ctx.accounts.admin.key(), ctx.accounts.config.admin,
        ErrorCode::Unauthorized);
    require!(!ctx.accounts.config.paused, ErrorCode::AlreadyPaused);  //
        State validation

    ctx.accounts.config.paused = true;
}
```

## IMPACT:

Redundant calls to enable_vrf succeed unnecessarily, creating inconsistent behavior with other state management functions.

## RECOMMENDATIONS:

Add state validation to prevent redundant calls:

```
pub fn enable_vrf(ctx: Context<UpdateConfig>) -> Result<()> {
    require_keys_eq!(ctx.accounts.admin.key(), ctx.accounts.config.admin,
        ErrorCode::Unauthorized);
    require!(!ctx.accounts.config.vrf_enabled,
        ErrorCode::VrfAlreadyEnabled);

    let cfg = &mut ctx.accounts.config;
    cfg.vrf_enabled = true;
}
```

# 4. CONCLUSION

In this audit, we thoroughly analyzed **DeFAI Swap & Stake** smart contract implementation. The problems found are described and explained in detail in Section 3. The problems found in the audit have been communicated to the project leader. We therefore consider the audit result to be **PASSED**.

To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

# 5. APPENDIX

## 5.1 Basic Coding Assessment

### 5.1.1 Apply Verification Control

| | |
|---|---|
| **Description** | The security of apply verification |
| **Result** | Not found |
| **Severity** | CRITICAL |

### 5.1.2 Authorization Access Control

| | |
|---|---|
| **Description** | Permission checks for external integral functions |
| **Result** | Not found |
| **Severity** | CRITICAL |

### 5.1.3 Forged Transfer Vulnerability

| | |
|---|---|
| **Description** | Assess whether there is a forged transfer notification vulnerability in the contract |
| **Result** | Not found |
| **Severity** | CRITICAL |

### 5.1.4 Transaction Rollback Attack

| | |
|---|---|
| **Description** | Assess whether there is transaction rollback attack vulnerability in the contract |
| **Result** | Not found |
| **Severity** | CRITICAL |

### 5.1.5 Transaction Block Stuffing Attack

| | |
|---|---|
| **Description** | Assess whether there is transaction blocking attack vulnerability |
| **Result** | Not found |
| **Severity** | CRITICAL |

### 5.1.6 Soft Fail Attack Assessment

| | |
|---|---|
| **Description** | Assess whether there is soft fail attack vulnerability |
| **Result** | Not found |
| **Severity** | CRITICAL |

### 5.1.7 Hard Fail Attack Assessment

| | |
|---|---|
| **Description** | Examine for hard fail attack vulnerability |
| **Result** | Not found |
| **Severity** | CRITICAL |

### 5.1.8 Abnormal Memo Assessment

| | |
|---|---|
| **Description** | Assess whether there is abnormal memo vulnerability in the contract |
| **Result** | Not found |
| **Severity** | CRITICAL |

### 5.1.9 Abnormal Resource Consumption

| | |
|---|---|
| **Description** | Examine whether abnormal resource consumption in contract processing |
| **Result** | Not found |
| **Severity** | CRITICAL |

### 5.1.10 Random Number Security

| | |
|---|---|
| **Description** | Examine whether the code uses insecure random number |
| **Result** | Not found |
| **Severity** | CRITICAL |

## 5.2 Advanced Code Scrutiny

### 5.2.1 Cryptography Security

| | |
|---|---|
| **Description** | Examine for weakness in cryptograph implementation |
| **Result** | Not found |
| **Severity** | HIGH |

### 5.2.2 Account Permission Control

| | |
|---|---|
| **Description** | Examine permission control issue in the contract |
| **Result** | Not found |
| **Severity** | MEDIUM |

### 5.2.3 Malicious Code Behavior

| | |
|---|---|
| **Description** | Examine whether sensitive behavior present in the code |
| **Result** | Not found |
| **Severity** | MEDIUM |

### 5.2.4 Sensitive Information Disclosure

| | |
|---|---|
| **Description** | Examine whether sensitive information disclosure issue present in the code |
| **Result** | Not found |
| **Severity** | MEDIUM |

### 5.2.5 System API

| | |
|---|---|
| **Description** | Examine whether system API application issue present in the code |
| **Result** | Not found |
| **Severity** | LOW |

## 6. DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without ExVul's prior written consent.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts ExVul to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. ExVul's position is that each company and individual are responsible for their own due diligence and continuous security. ExVul's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

## 7. REFERENCES

[1] MITRE. CWE-191: Integer Underflow (Wrap or Wraparound). https://cwe.mitre.org/data/definitions/191.html.

[2] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html.

[3] MITRE. CWE-400: Uncontrolled Resource Consumption. https://cwe.mitre.org/data/definitions/400.html.

[4] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440.html.

[5] MITRE. CWE-684: Protection Mechanism Failure. https://cwe.mitre.org/data/definitions/693.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE CATEGORY: Resource Management Errors. https://cve.mitre.org/data/definitions/399.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

# Contact

🌐 **Website**
**www.exvul.com**

✉ **Email**
**contact@exvul.com**

𝕏 **Twitter**
**@EXVULSEC**

**Github**
**github.com/EXVUL-Sec**

ExVul