SMART CONTRACT AUDIT REPORT

DefAI Swap Smart Contract

AUGUST 2025

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **DefAI Swap** 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 \ IMPACT	Informational	Low	Medium	High
High	INFO	MEDIUM	HIGH	CRITICAL
Medium	INFO	LOW	MEDIUM	HIGH
Low	INFO	LOW	LOW	MEDIUM

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 • Authorization Access Control • Forged Transfer Vulnerability • Forged Transfer Notification • Numeric Overflow • Transaction Rollback Attack • Transaction Block Stuffing Attack • Soft Fail Attack • Hard Fail Attack • Abnormal Memo • Abnormal Resource Consumption • Secure Random Number

Advanced Source Code Scrutiny	• Asset Security • Cryptography Security • Business Logic Review • Source Code Functional Verification • Account Authorization Control • Sensitive Information Disclosure • Circuit Breaker • Blacklist Control • System API Call Analysis • Contract Deployment Consistency Check • Abnormal Resource Consumption
Additional Recommendations	• Semantic Consistency Checks • 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

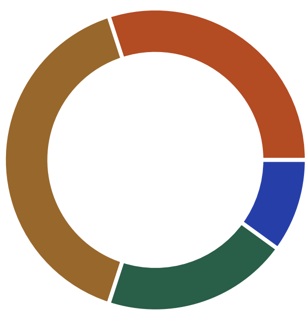
Project Name	Audit Time	Language
DefAI	21/07/2025 - 11/08/2025	Rust

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

Commit Hash
bbf88147743821d60bdbcf335e25e8ac9159f441

2.2 Summary

Severity	Found
CRITICAL	0
HIGH	3
MEDIUM	4
LOW	2
INFO	1



2.3 Key Findings

Severity	Findings Title	Status
HIGH	Missing treasury account validation allows tax funds to be redirected to arbitrary accounts	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	OG Tier 0 and paid Tier 0 share supply pool allowing paid users to exhaust OG allocation	Fixed
MEDIUM	Tax reset time comparison inconsistency creates unfair economic advantage	Fixed
MEDIUM	Pause mechanism not enforced in critical functions	Fixed
MEDIUM	VRF switch and results not being used	Fixed
LOW	Missing admin authorization validation in whitelist initialization	Fixed
LOW	OLD DEFAI swap breaks tax reset mechanism causing unfair tax burden	Fixed
INFO	Enable VRF function lacks state validation allowing redundant calls	Fixed

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Missing treasury account validation allows tax funds to be redirected to arbitrary accounts

SEVERITY: HIGH **STATUS:** Fixed

PATH:

security-auditor/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.

The vulnerable code section:

```
#[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>>,
    /// CHECK: DEFAI mint
    pub defai_mint: AccountInfo<'info>,
    pub config: Account<'info, Config>,
    // ...
}

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)?;
}
```

The CollectionConfig struct contains the real treasury address, but the function doesn't validate that the provided treasury_defai_ata matches the configured address.

IMPACT:

Tax funds can be redirected to attacker-controlled accounts instead of the legitimate treasury. This allows attackers to steal protocol revenue while the legitimate treasury receives no funds.

RECOMMENDATIONS:

Add validation to ensure treasury and escrow accounts match the configured addresses:

```
#[derive(Accounts)]
pub struct SwapDefaiForPnftV6<'info> {
    #[account(
        mut,
        constraint = treasury_defai_ata.owner ==
           collection_config.treasury
            @ ErrorCode::InvalidTreasury
    )]
    pub treasury_defai_ata: Box<InterfaceAccount<'info,
       TokenAccount2022>>,

    #[account(
        mut,
        seeds = [b"escrow"],
        bump,
        constraint = escrow_defai_ata.owner == escrow.key()
            @ ErrorCode::InvalidEscrow
```

```
    )]
    pub escrow_defai_ata: Box<InterfaceAccount<'info, TokenAccount2022>>,

    pub collection_config: Box<Account<'info, CollectionConfig>>,
    // ...
}
```

3.2 Missing NFT mint validation allows unauthorized vesting claims

SEVERITY: HIGH **STATUS:** Fixed

PATH:

`security-auditor/defai_swap/src/lib.rs::claim_vested_v6 and 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.

The vulnerable code section:

```
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
    );
    // ...
}
```

The functions validate ownership and amount but not the mint address, allowing users to use any NFT they own to claim rewards for a different NFT's vesting state.

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. This allows unauthorized access to vesting rewards that should only be available to specific NFT holders.

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
    );

    let vesting_state = &mut ctx.accounts.vesting_state;
    // ...
}

pub fn reroll_bonus_v6(ctx: Context<RerollBonusV6>) -> 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
    );
    // ...
}
```

3.3 Token-2022 Funds Permanently Locked Due to AdminWithdraw Token Standard Mismatch

SEVERITY: HIGH **STATUS:** Fixed

PATH:

security-auditor/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.

Token Standard Inconsistency:

```
// 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>>,
}

// Uses Token-2022 transfer
let cpi_ctx_tax = CpiContext::new_with_signer(
    ctx.accounts.token_program_2022.to_account_info(),
    TransferChecked {
        from: ctx.accounts.user_defai_ata.to_account_info(),
        to: ctx.accounts.escrow_defai_ata.to_account_info(),
        authority: ctx.accounts.user.to_account_info(),
        mint: ctx.accounts.defai_mint.to_account_info(),
    },
    &[&user_seeds[..]],
);
token22::transfer_checked(cpi_ctx_tax, tax_amount, 6)?;
```

But admin_withdraw only supports standard SPL Token:

```
// AdminWithdraw only supports standard SPL Token
#[derive(Accounts)]
pub struct AdminWithdraw<'info> {
    #[account(mut)]
    pub source_vault: Account<'info, TokenAccount>,
    #[account(mut)]
    pub dest: 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)?;  // Standard SPL Token transfer
}
```

IMPACT:

All DEFAI fees from swap_defai_for_pnft_v6 are permanently locked and cannot be withdrawn by the admin. This results in complete loss of protocol revenue from Token-2022 based swaps.

RECOMMENDATIONS:

Add Token-2022 withdrawal support to the admin_withdraw function:

```
pub fn admin_withdraw(
    ctx: Context<AdminWithdraw>,
    amount: u64,
    is_token2022: bool,
) -> Result<()> {
    require_keys_eq!(ctx.accounts.admin.key(), ctx.accounts.config.admin,
```

```
        ErrorCode::Unauthorized);

    if is_token2022 {
        // Token-2022 withdrawal logic
        let cpi_ctx = CpiContext::new_with_signer(
            ctx.accounts.token_program_2022.to_account_info(),
            TransferChecked {
                from: ctx.accounts.source_vault_2022.to_account_info(),
                to: ctx.accounts.dest_2022.to_account_info(),
                authority: ctx.accounts.escrow.to_account_info(),
                mint: ctx.accounts.mint.to_account_info(),
            },
            &[&escrow_seeds[..]],
        );
        token22::transfer_checked(cpi_ctx, amount, 6)?;
    } else {
        // Standard SPL Token withdrawal logic
        let cpi_ctx = CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            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)?;
    }

    Ok(())
}
```

3.4 OG Tier 0 and paid Tier 0 share supply pool allowing paid users to exhaust OG allocation

SEVERITY: MEDIUM

STATUS: Fixed

PATH:

security-auditor/defai_swap/src/lib.rs::swap_og_tier0_for_pnft_v6 and 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.

The vulnerable code sections:

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

    config.tier_minted[0] += 1;  // Same increment
}

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;  // For tier=0, same increment
```

```
}
```

Both functions check and increment the same tier_minted[0] counter against the same tier_supplies[0] limit, creating a race condition where paid users can exhaust the supply before OG holders claim their reserved allocation.

IMPACT:

Paid users can exhaust the Tier 0 supply before OG Tier 0 holders claim their reserved allocation. This results in OG Tier 0 holders being unable to claim their free NFTs due to supply exhaustion, creating an 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:

```
#[account]
pub struct CollectionConfig {
    // ...
    pub tier_supplies: [u16; 5],
    pub tier_minted: [u16; 5],
    pub og_tier_0_supply: u16,      // Reserved supply for OG holders
    pub og_tier_0_minted: u16,      // Counter for OG claims
    // ...
}

pub fn swap_og_tier0_for_pnft_v6(ctx: Context<SwapOgTier0ForPnftV6>, ...)
    -> Result<()> {
    require!(config.og_tier_0_minted < config.og_tier_0_supply,
        ErrorCode::NoLiquidity);

    config.og_tier_0_minted += 1;
}

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

    // For tier 0, check remaining supply after reserving for OG holders
    if tier == 0 {
```

```
        let remaining_supply =
            config.tier_supplies[0].saturating_sub(config.og_tier_0_supply);
        require!(config.tier_minted[0] < remaining_supply,
            ErrorCode::NoLiquidity);
    } else {
        require!(config.tier_minted[tier as usize] <
            config.tier_supplies[tier as usize], ErrorCode::NoLiquidity);
    }

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

3.5 Tax reset time comparison inconsistency creates unfair economic advantage

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`security-auditor/defai_swap/src/lib.rs::reset_user_tax and swap_defai_for_pnft_v`

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.

The inconsistent code sections:

```
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;
    }

    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;
}
```

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. This creates an unfair 25% tax difference for identical timing conditions.

RECOMMENDATIONS:

Standardize time comparison logic to use >= in both functions:

```
pub fn swap_defai_for_pnft_v6(ctx: Context<SwapDefaiForPnftV6>, tier: u8,
    ...) -> Result<()> {
    // Use >= to match reset_user_tax logic
    if clock.unix_timestamp - user_tax.last_swap_timestamp >=
        TAX_RESET_DURATION {
        user_tax.tax_rate_bps = INITIAL_TAX_BPS;
        user_tax.swap_count = 0;
    }

    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;

    // Rest of the function...
}
```

3.6 Pause mechanism not enforced in critical functions

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`security-auditor/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.

The vulnerable functions lack pause checks:

```
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;
    // ... rest of swap logic
}

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;
    // ... rest of claim logic
}

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

While the contract has pause/unpause functionality:

```
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);

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

IMPACT:

Protocol operations continue even when admin has paused the system. The emergency pause mechanism is ineffective for critical functions, allowing users to still swap, claim, and redeem during emergency situations when the protocol should be halted.

RECOMMENDATIONS:

Add pause validation to all critical functions:

```
pub fn swap_defai_for_pnft_v6(ctx: Context<SwapDefaiForPnftV6>, tier: u8,
    ...) -> Result<()> {
    // Add pause check at the beginning
    require!(!ctx.accounts.config.paused, ErrorCode::ProtocolPaused);

    let config = &mut ctx.accounts.collection_config;
    // ... rest of swap logic
}

pub fn claim_vested_v6(ctx: Context<ClaimVestedV6>) -> Result<()> {
    // Add pause check at the beginning
    require!(!ctx.accounts.config.paused, ErrorCode::ProtocolPaused);

    let vesting_state = &mut ctx.accounts.vesting_state;
    // ... rest of claim logic
}

pub fn redeem_v6(ctx: Context<RedeemV6>) -> Result<()> {
    // Add pause check at the beginning
    require!(!ctx.accounts.config.paused, ErrorCode::ProtocolPaused);

    let bonus_state = &mut ctx.accounts.bonus_state;
    // ... rest of redeem logic
}
```

3.7 VRF switch and results not being used

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`security-auditor/defai_swap/src/lib.rs::enable_vrf and 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.

The problematic implementation:

```
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<()> {
    // VRF flag is ignored, always uses weak randomness
    let random_value = generate_secure_random(
        &ctx.accounts.user.key(),
        &ctx.accounts.nft_mint.key(),
        &clock,
        &blockhash_bytes,
    );

    // VRF infrastructure exists but result_buffer is never used
    // ...
}
```

The VRF infrastructure exists but the actual VRF results are never consumed by the randomness generation functions.

IMPACT:

VRF functionality is effectively disabled despite being "enabled". Core functions continue using predictable randomness from recent_blockhashes, and the VRF infrastructure exists but provides no security benefit. This defeats the purpose of implementing VRF for secure randomness.

RECOMMENDATIONS:

Enforce VRF usage when enabled and use VRF results:

```
pub fn swap_defai_for_pnft_v6(ctx: Context<SwapDefaiForPnftV6>, tier: u8,
    ...) -> Result<()> {
    // Check VRF enabled status and use appropriate randomness
    let random_value = if ctx.accounts.config.vrf_enabled {
        // Require VRF result to be ready
        require!(ctx.accounts.vrf_state.result_buffer != [0u8; 32],
            ErrorCode::VrfNotReady);

        // Use VRF result for secure randomness
        generate_vrf_random(
            &ctx.accounts.vrf_state.result_buffer,
            &ctx.accounts.user.key(),
            &ctx.accounts.nft_mint.key(),
        )
    } else {
        // Use weak randomness only when VRF disabled
        generate_secure_random(
            &ctx.accounts.user.key(),
            &ctx.accounts.nft_mint.key(),
            &clock,
            &blockhash_bytes,
        )
    };

    // Use random_value for bonus calculations...
}

fn generate_vrf_random(
    vrf_result: &[u8; 32],
    user_key: &Pubkey,
    nft_mint: &Pubkey,
) -> u64 {
    // Combine VRF result with user-specific data for deterministic
        randomness
```

```
    let mut hasher = std::collections::hash_map::DefaultHasher::new();
    hasher.write(vrf_result);
    hasher.write(user_key.as_ref());
    hasher.write(nft_mint.as_ref());
    hasher.finish()
}
```

3.8 Missing admin authorization validation in whitelist initialization

SEVERITY: LOW **STATUS:** Fixed

PATH:

security-auditor/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.

The vulnerable code:

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

#[derive(Accounts)]
pub struct InitializeWhitelist<'info> {
     #[account(mut)]
     pub admin: Signer<'info>,  // No validation for admin authority
     #[account(
         init,
         payer = admin,
         space = 8 + Whitelist::LEN,
         seeds = [b"whitelist"],
         bump,
     )]
     pub whitelist: Account<'info, Whitelist>,
     pub system_program: Program<'info, System>,
}
```

IMPACT:

Any user can initialize the whitelist by providing a signature and paying account creation fees. While the impact is limited since the function can only be called once due to PDA constraints, it still represents improper access control.

RECOMMENDATIONS:

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

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

    #[account(
        seeds = [b"config"],
        bump
    )]
    pub config: Account<'info, Config>,

    #[account(
        init,
        payer = admin,
        space = 8 + Whitelist::LEN,
        seeds = [b"whitelist"],
        bump,
    )]
    pub whitelist: Account<'info, Whitelist>,
    pub system_program: Program<'info, System>,
}

pub fn initialize_whitelist(ctx: Context<InitializeWhitelist>) ->
    Result<()> {
    // Validate admin authority
    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(())
}
```

3.9 OLD DEFAI swap breaks tax reset mechanism causing unfair tax burden

SEVERITY: LOW **STATUS:** Fixed

PATH:

security-auditor/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.

The problematic code:

```
pub fn swap_old_defai_for_pnft_v6(
    ctx: Context<SwapOldDefaiForPnftV6>,
    tier: u8,
    _metadata_uri: String,
    _name: String,
    _symbol: String,
) -> Result<()> {
    // ... no tax charged for OLD DEFAI

    // Update user tax state - breaks reset mechanism
    user_tax.swap_count += 1;
    user_tax.last_swap_timestamp = clock.unix_timestamp;  // This breaks
       tax reset
}
```

The function lacks the tax reset logic that exists in swap_defai_for_pnft_v6:

```
// swap_defai_for_pnft_v6 has tax reset logic
if clock.unix_timestamp - user_tax.last_swap_timestamp >
   TAX_RESET_DURATION {
    user_tax.tax_rate_bps = INITIAL_TAX_BPS;
    user_tax.swap_count = 0;
}
```

```
// swap_old_defai_for_pnft_v6 lacks this check
// Only updates timestamp, breaking the reset mechanism
```

IMPACT:

Users using OLD DEFAI are penalized with higher taxes when they later use DEFAI tokens. This creates a logic contradiction where "no tax" transactions still affect the tax state, preventing the 24-hour tax reset mechanism from working properly.

RECOMMENDATIONS:

Add tax reset logic to swap_old_defai_for_pnft_v6 or remove the timestamp update entirely:

```
pub fn swap_old_defai_for_pnft_v6(
    ctx: Context<SwapOldDefaiForPnftV6>,
    tier: u8,
    _metadata_uri: String,
    _name: String,
    _symbol: String,
) -> Result<()> {
    // ... existing logic

    // Add tax reset logic similar to swap_defai_for_pnft_v6
    if clock.unix_timestamp - user_tax.last_swap_timestamp >
        TAX_RESET_DURATION {
         user_tax.tax_rate_bps = INITIAL_TAX_BPS;
         user_tax.swap_count = 0;
    }

    // Update user tax state
    user_tax.swap_count += 1;
    user_tax.last_swap_timestamp = clock.unix_timestamp;

    // OR alternatively, don't update timestamp for OLD DEFAI swaps:
    // user_tax.swap_count += 1;
    // // Don't update timestamp for OLD DEFAI since no tax affects tax
        calculation
}
```

3.10 Enable VRF function lacks state validation allowing redundant calls

SEVERITY: INFO **STATUS:** Fixed

PATH:

security-auditor/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.

The vulnerable code:

```
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 the pause function which properly validates state:

```
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);  //
        Proper validation
```

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

IMPACT:

Redundant calls to enable_vrf succeed unnecessarily, creating inconsistent behavior with other state management functions and potentially causing confusion in admin operations and event logs.

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);

    // Add state validation to prevent redundant calls
    require!(!ctx.accounts.config.vrf_enabled,
        ErrorCode::VrfAlreadyEnabled);

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

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

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

    Ok(())
}

// Add corresponding error code
#[error_code]
pub enum ErrorCode {
    // ... existing error codes
    #[msg("VRF is already enabled")]
    VrfAlreadyEnabled,
```

```
}
```

4. CONCLUSION

In this audit, we thoroughly analyzed **DefAI Swap** 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://cwe.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

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec

ExVul