Cyberscope

Audit Report

GroWealth

December 2024

Source	Github Repository
Commit	a5013e8bad95caedf0150b876081ad35ba0c6007
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope’s smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Commit	a5013e8bad95caedf0150b876081ad35ba0c6007
Network	SOL

Audit Updates

Initial Audit	05 Nov 2024
Corrected Phase 2	19 Dec 2024

Source Files

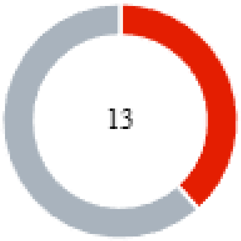
Filename	SHA256
constant.rs	922cbbafae3888d664cd953d6b8efa7a53a6507ec05a63aeddb129dba7bf369a
error.rs	4167b4ba7a82a85f01f22a8390df916039c104e2afe7375286975bd161284236
events.rs	9b5b535eb052f1cfd2401db6cdeea0b73f1885ea184217c2e1ccd7e714feb907
lib.rs	bf36aeb2434919e4c6ecafea3711772ce9e595eb2ee1553a4e8f6ffda8711f6c
state.rs	506b626ed5fd7e07c8152ec57f12da5de3e9318c8ecf2a2799077f79396936ba
processor/create_presale.rs	f12eb4acb06be1fa5eadd8d1c433919e433c218d0047d8e4ba397d4be91be9e2
processor/grant_access.rs	65493d1debf4322281dfd601a0aa9c20229151d9b222752ebd0e26a19307019c

processor/initialize.rs	3116e5b9dbf13c2658c82c19fb8091cbed19c2f3cbefdcd1fe9f95e13d75f7c6
processor/mod.rs	abedce22b9ba191e7aa5fd28c7fd74ab137190a2240590ba8a08ed2bdf27620c
processor/purchase_token.rs	a4b0c95194ec4b4f8d59b768a9e1e2f5091b7f37b54d572895a39169a7214d4a
processor/revoke.rs	f468592f72b5dfb707c9a0d795fcf207dc5dcc26bfce42689526cf4efe79aa83
processor/update_presale.rs	82453cc2b063558fb75b369287d543fc693aa706e578c0fa584b15a02114a43f
processor/withdraw_token.rs	91a85e93bbb528c7135126a1135a1d2cef00edbc6b64cb666079835843a1bacc

Contract Readability Comment

The audit scope is to check for security vulnerabilities, validate the business logic, and propose potential optimizations. The contract does not adhere to key principles of Solana and Rust development regarding resource efficiency, code readability, and proper data structures. Given these issues, it cannot be considered production-ready. The development team is strongly advised to re-evaluate the business logic and follow Solana and Rust best practices. It is recommended to optimize resource usage, simplify function definitions, and use descriptive variable names to enhance auditability, efficiency, and security.

Findings Breakdown



●	Critical	5
●	Medium	0
●	Minor / Informative	8

Severity	Unresolved	Acknowledged	Resolved	Other
● Critical	5	0	0	0
● Medium	0	0	0	0
● Minor / Informative	8	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	PAR	Potential Arbitrage Risk	Unresolved
🔴	CAVA	Creator Account Validation Absence	Unresolved
🔴	FI	Frontunnable Initialization	Unresolved
🔴	IPMV	Inadequate Payment Mint Validation	Unresolved
🔴	IDH	Incorrect Decimal Handling	Unresolved
⚪	RCC	Redundant Conditional Check	Unresolved
⚪	IKCU	Inconsistent Key Comparison Use	Unresolved
⚪	PPR	Potential Presale Reinitialization	Unresolved
⚪	PCR	Program Centralization Risk	Unresolved
⚪	RFN	Redundant Field Names	Unresolved
⚪	SSI	Single Seed Initialization	Unresolved
⚪	UAIC	Unused Account in Context	Unresolved
⚪	UEC	Unused Error Codes	Unresolved

PAR - Potential Arbitrage Risk

Criticality	Critical
Location	processor/purchase_token.rs#L21
Status	Unresolved

Description

In the current implementation of the presale, users receive their purchased tokens immediately upon successful payment. This creates a vulnerability where users can exploit arbitrage opportunities by using the tokens received from the presale to create liquidity pools or pairs on decentralized exchanges, such as Raydium, at prices that differ from the fixed price of the presale. For instance, a user could purchase tokens at the presale's fixed price and immediately create a trading pair on Raydium at a higher price. If other market participants trade within the newly created pool, the user can profit from the arbitrage between the presale price and the pool price. This disrupts the intended tokenomics and market stability, particularly when the token's market price diverges significantly from the presale price.

```
pub fn purchase_token_handler(ctx: Context<PurchaseToken>,
token_amount: u64) -> Result<()> {
    let clock = Clock::get()?;
    let current_unix_timestamp = clock.unix_timestamp as u64;

    // Payment (USDC) decimals
    let payment_decimal_value =
(10u64).pow(ctx.accounts.payment_mint.decimals as u32);

    // Presale token (Token-2022) decimals
    let token_decimal_value =
(10u64).pow(ctx.accounts.token_mint.decimals as u32);

    require!(
        current_unix_timestamp >=
ctx.accounts.presale_account.start_time &&
            current_unix_timestamp <=
ctx.accounts.presale_account.end_time,
        PresaleErrorCodes::InvalidTime
    );

    // Check token_amount is within the buyable range
    require!(
        token_amount >=
ctx.accounts.presale_account.minimum_buyable_amount &&
            token_amount <=
ctx.accounts.presale_account.maximum_buyable_amount,
        PresaleErrorCodes::InvalidPurchaseAmount
    );

    // Calculate the USDC payment amount
    let payment_amount =
        ctx.accounts.presale_account.token_price_in_usdc *
(token_amount / token_decimal_value);
    msg!("Payment amount: {}", (payment_amount as f64) /
(payment_decimal_value as f64));

    // Ensure payment_amount is greater than 0
    require!(payment_amount > 0, PresaleErrorCodes::ZeroPaymentAmount);

    if payment_amount > 0 {
        // Transfer USDC payment from buyer to creator
        anchor_spl::token::transfer_checked(
            ctx.accounts.transfer_usdc_payment_to_creator(),
            payment_amount,
            ctx.accounts.payment_mint.decimals
        )?;
        msg!("Payment Done");

        // Now transfer the presale tokens (Token-2022) from presale to
buyer
        let bump = ctx.accounts.presale_account.bump;
        anchor_spl::token_interface::transfer_checked(

ctx.accounts.transfer_presale_token_to_buyer().with_signer(&[&[PRESALE_
SEED, &[bump]]]),
            token_amount,
            ctx.accounts.token_mint.decimals
        )?;
        msg!("Token bought successfully");

        ctx.accounts.presale_account.total_tokens -= token_amount;
    } else {
    }

    emit!(PurchaseTokenEvent {
        buyer: ctx.accounts.buyer.key(),
        bought_token_amount: token_amount,
    });
    Ok(())
}
```

Recommendation

The presale mechanism should be adjusted so that users do not receive tokens instantly. Instead, token distribution should occur only after the presale has finalized. This can be enforced through a finalization mechanism, such as a dedicated function callable by the contract owner or an automatic trigger once the presale period ends. Delaying the token distribution until finalization prevents users from immediately using presale tokens for arbitrage and helps maintain the presale’s intended price stability.

CAVA - Creator Account Validation Absence

Criticality	Critical
Location	processor/purchase_token.rs#L21,82
Status	Unresolved

Description

The `purchase_token_handler` or the `PurchaseToken` does not validate that the creator's USDC receiving associated token account (ATA) belongs to the intended creator. This allows a malicious actor to provide an arbitrary ATA instead of the legitimate one, potentially redirecting the payment to an unauthorized account. As a result, the funds meant for the creator could be diverted or even returned to the user themselves, undermining the integrity of the presale process.

```
pub fn purchase_token_handler(ctx: Context<PurchaseToken>,
token_amount: u64) -> Result<()> {
    let clock = Clock::get()?;
    let current_unix_timestamp = clock.unix_timestamp as u64;

    // Payment (USDC) decimals
    let payment_decimal_value =
(10u64).pow(ctx.accounts.payment_mint.decimals as u32);

    // Presale token (Token-2022) decimals
    let token_decimal_value =
(10u64).pow(ctx.accounts.token_mint.decimals as u32);

    require!(
        current_unix_timestamp >=
ctx.accounts.presale_account.start_time &&
            current_unix_timestamp <=
ctx.accounts.presale_account.end_time,
        PresaleErrorCodes::InvalidTime
    );

    // Check token_amount is within the buyable range
    require!(
        token_amount >=
ctx.accounts.presale_account.minimum_buyable_amount &&
            token_amount <=
ctx.accounts.presale_account.maximum_buyable_amount,
        PresaleErrorCodes::InvalidPurchaseAmount
    );

    // Calculate the USDC payment amount
    let payment_amount =
        ctx.accounts.presale_account.token_price_in_usdc *
(token_amount / token_decimal_value);
    msg!("Payment amount: {}", (payment_amount as f64) /
(payment_decimal_value as f64));

    // Ensure payment_amount is greater than 0
    require!(payment_amount > 0, PresaleErrorCodes::ZeroPaymentAmount);

    if payment_amount > 0 {
        // Transfer USDC payment from buyer to creator
        anchor_spl::token::transfer_checked(
            ctx.accounts.transfer_usdc_payment_to_creator(),
            payment_amount,
            ctx.accounts.payment_mint.decimals
        )?;
        msg!("Payment Done");

        // Now transfer the presale tokens (Token-2022) from presale to
buyer
        let bump = ctx.accounts.presale_account.bump;
        anchor_spl::token_interface::transfer_checked(

ctx.accounts.transfer_presale_token_to_buyer().with_signer(&[&[PRESALE_
SEED, &[bump]]]),
            token_amount,
            ctx.accounts.token_mint.decimals
        )?;
        msg!("Token bought successfully");

        ctx.accounts.presale_account.total_tokens -= token_amount;
    } else {
    }

    emit!(PurchaseTokenEvent {
        buyer: ctx.accounts.buyer.key(),
        bought_token_amount: token_amount,
    });
    Ok(())
}

#[derive(Accounts)]
pub struct PurchaseToken<'info> {
    #[account(mut)]
    pub fee_payer: Signer<'info>,

    /// CHECK: Transaction will be signed by the buyer externally
(front-end)
    #[account(mut, signer)]
    pub buyer: AccountInfo<'info>,

    #[account(mut, seeds = [PRESALE_SEED], bump)]
    pub presale_account: Box<Account<'info, PresaleAccount>>,

    // The buyer receives the Token-2022 token in this account
    #[account(
        init_if_needed,
        payer = fee_payer,
        associated_token::mint = token_mint,
        associated_token::authority = buyer
    )]
    pub buyer_token_ata: Box<InterfaceAccount<'info,
TokenAccount2022>>,

    // Buyer's USDC payment account (original token)
    #[account(mut)]
    pub buyer_payment_ata: Account<'info, TokenAccountLegacy>,

    // Presale's token-2022 source account
    #[account(mut)]
    pub presale_token_ata: Box<InterfaceAccount<'info,
TokenAccount2022>>,

    #[account(
        mut,
        seeds = [PRESALE_SEED, PROGRAM_DATA_SEED],
        bump = presale_program_data.bump
    )]
    pub presale_program_data: Box<Account<'info, PresaleProgramData>>,

    // Token-2022 mint for the presale token
    #[account(
        mut,
        constraint = token_mint.key().as_ref() ==
presale_program_data.token_mint.as_ref()
            @ PresaleErrorCodes::InvalidMintedToken
    )]
    pub token_mint: Box<InterfaceAccount<'info, Mint2022>>,

    // USDC payment mint (original token)
    #[account(mut)]
    pub payment_mint: Account<'info, MintLegacy>,

    #[account(
        mut,
        seeds = [CREATOR_SEED, creator_account.creator.key().as_ref()],
        bump
    )]
    pub creator_account: Box<Account<'info, CreatorAccount>>,

    // Creator USDC receiving ATA (original token)
    #[account(mut)]
    pub creator_payment_token_ata: Account<'info, TokenAccountLegacy>,

    // Program for Token-2022 transfers (presale token)
    pub token_program: Program<'info, Token2022>,

    // Program for Associated Token Accounts
    pub associated_token_program: Program<'info, AssociatedToken>,

    // Program for system account
    pub system_program: Program<'info, System>,

    // Rent sysvar
    pub rent: Sysvar<'info, Rent>,

    // ADD the original token program for USDC transfers
    #[account(address = anchor_spl::token::ID)]
    pub token_program_usdc: Program<'info, TokenLegacy>,
}
```

Recommendation

The smart contract should enforce a strict validation to ensure that the provided ATA for receiving USDC payments corresponds to the actual creator. This prevents unauthorized redirection of funds and ensures that the creator receives the intended payments. Strengthening this validation protects the integrity of the presale and secures the transaction flow.

FI - Frontunnable Initialization

Criticality	Critical
Location	processor/initialize.rs#L12
Status	Unresolved

Description

The `initialize_handler` function lacks any access control mechanism, allowing any account to call it. This function sets the caller as the authority of the `presale_program_data` account, granting them full administrative control over the presale configuration. Without restrictions, any user can initialize the presale, taking ownership and potentially compromising the intended control of the contract.

```
pub fn initialize_handler(ctx: Context<Initialize>, args:
InitializeArgs) -> Result<()> {
    let presale_program_data: &mut Account<PresaleProgramData> = &mut
ctx.accounts.presale_program_data;
    presale_program_data.token_mint = args.token_mint;
    presale_program_data.super_authority =
ctx.accounts.authority.key();
    presale_program_data.bump = ctx.bumps.presale_program_data;

    emit!(InitializeEvent {
        initializer: ctx.accounts.authority.key(),
        token_mint: args.token_mint,
    });
    Ok(())
}
```

Recommendation

Add access control to the `initialize_handler` function to ensure that only an authorized entity can call it. Consider using the program's upgrade authority as the designated initializer or specify a hardcoded address of a trusted account. This change prevents unauthorized users from taking control and secures the initialization process.

IPMV - Inadequate Payment Mint Validation

Criticality	Critical
Location	processor/purchase_token.rs#L21,82
Status	Unresolved

Description

The contract currently relies on the official SPL token program address to validate the token program used for payments, but does not verify that the provided payment mint address belongs to the intended stable asset. This allows a malicious actor to supply a different SPL token mint instead of the legitimate stable token mint, potentially causing the contract to accept undesirable or worthless tokens as payment. By not enforcing that the payment mint is the expected stablecoin, the contract leaves itself vulnerable to manipulation and exploitation.

```
pub fn purchase_token_handler(ctx: Context<PurchaseToken>,
token_amount: u64) -> Result<()> {
    let clock = Clock::get()?;
    let current_unix_timestamp = clock.unix_timestamp as u64;

    // Payment (USDC) decimals
    let payment_decimal_value =
(10u64).pow(ctx.accounts.payment_mint.decimals as u32);

    // Presale token (Token-2022) decimals
    let token_decimal_value =
(10u64).pow(ctx.accounts.token_mint.decimals as u32);

    require!(
        current_unix_timestamp >=
ctx.accounts.presale_account.start_time &&
            current_unix_timestamp <=
ctx.accounts.presale_account.end_time,
        PresaleErrorCodes::InvalidTime
    );

    // Check token_amount is within the buyable range
    require!(
        token_amount >=
ctx.accounts.presale_account.minimum_buyable_amount &&
            token_amount <=
ctx.accounts.presale_account.maximum_buyable_amount,
        PresaleErrorCodes::InvalidPurchaseAmount
    );

    // Calculate the USDC payment amount
    let payment_amount =
        ctx.accounts.presale_account.token_price_in_usdc *
(token_amount / token_decimal_value);
    msg!("Payment amount: {}", (payment_amount as f64) /
(payment_decimal_value as f64));

    // Ensure payment_amount is greater than 0
    require!(payment_amount > 0, PresaleErrorCodes::ZeroPaymentAmount);

    if payment_amount > 0 {
        // Transfer USDC payment from buyer to creator
        anchor_spl::token::transfer_checked(
            ctx.accounts.transfer_usdc_payment_to_creator(),
            payment_amount,
            ctx.accounts.payment_mint.decimals
        )?;
        msg!("Payment Done");

        // Now transfer the presale tokens (Token-2022) from presale to
buyer
        let bump = ctx.accounts.presale_account.bump;
        anchor_spl::token_interface::transfer_checked(

ctx.accounts.transfer_presale_token_to_buyer().with_signer(&[&[PRESALE_
SEED, &[bump]]]),
            token_amount,
            ctx.accounts.token_mint.decimals
        )?;
        msg!("Token bought successfully");

        ctx.accounts.presale_account.total_tokens -= token_amount;
    } else {
    }

    emit!(PurchaseTokenEvent {
        buyer: ctx.accounts.buyer.key(),
        bought_token_amount: token_amount,
    });
    Ok(())
}

#[derive(Accounts)]
pub struct PurchaseToken<'info> {
    #[account(mut)]
    pub fee_payer: Signer<'info>,

    /// CHECK: Transaction will be signed by the buyer externally
(front-end)
    #[account(mut, signer)]
    pub buyer: AccountInfo<'info>,

    #[account(mut, seeds = [PRESALE_SEED], bump)]
    pub presale_account: Box<Account<'info, PresaleAccount>>,

    // The buyer receives the Token-2022 token in this account
    #[account(
        init_if_needed,
        payer = fee_payer,
        associated_token::mint = token_mint,
        associated_token::authority = buyer
    )]
    pub buyer_token_ata: Box<InterfaceAccount<'info,
TokenAccount2022>>,

    // Buyer's USDC payment account (original token)
    #[account(mut)]
    pub buyer_payment_ata: Account<'info, TokenAccountLegacy>,

    // Presale's token-2022 source account
    #[account(mut)]
    pub presale_token_ata: Box<InterfaceAccount<'info,
TokenAccount2022>>,

    #[account(
        mut,
        seeds = [PRESALE_SEED, PROGRAM_DATA_SEED],
        bump = presale_program_data.bump
    )]
    pub presale_program_data: Box<Account<'info, PresaleProgramData>>,

    // Token-2022 mint for the presale token
    #[account(
        mut,
        constraint = token_mint.key().as_ref() ==
presale_program_data.token_mint.as_ref()
            @ PresaleErrorCodes::InvalidMintedToken
    )]
    pub token_mint: Box<InterfaceAccount<'info, Mint2022>>,

    // USDC payment mint (original token)
    #[account(mut)]
    pub payment_mint: Account<'info, MintLegacy>,

    #[account(
        mut,
        seeds = [CREATOR_SEED, creator_account.creator.key().as_ref()],
        bump
    )]
    pub creator_account: Box<Account<'info, CreatorAccount>>,

    // Creator USDC receiving ATA (original token)
    #[account(mut)]
    pub creator_payment_token_ata: Account<'info, TokenAccountLegacy>,

    // Program for Token-2022 transfers (presale token)
    pub token_program: Program<'info, Token2022>,

    // Program for Associated Token Accounts
    pub associated_token_program: Program<'info, AssociatedToken>,

    // Program for system account
    pub system_program: Program<'info, System>,

    // Rent sysvar
    pub rent: Sysvar<'info, Rent>,

    // ADD the original token program for USDC transfers
    #[account(address = anchor_spl::token::ID)]
    pub token_program_usdc: Program<'info, TokenLegacy>,
}
```

Recommendation

The contract should include a validation step that ensures the provided payment mint address matches the known stable asset mint. This additional verification ensures that all payments come from the intended, trusted token mint and prevents attackers from substituting other tokens to subvert the payment process.

IDH - Incorrect Decimal Handling

Criticality	Critical
Location	processor/purchase_token.rs#L21
Status	Unresolved

Description

The calculation of the payment amount for purchasing tokens includes a division operation that truncates the decimal portion of the token amount due to the placement of parentheses. This results in users underpaying for the tokens they receive. Specifically, when a user attempts to purchase fractional tokens, the payment calculation only considers the truncated integer portion, but the user still receives the full fractional token amount. For instance, if a user intends to purchase 1.9 tokens, they pay only for 1 token in USDC due to truncation but receive 1.9 tokens. This discrepancy leads to a mismatch between the amount paid and the tokens received, potentially resulting in economic losses for the presale authority and undermining the integrity of the presale.

```
pub fn purchase_token_handler(ctx: Context<PurchaseToken>,
token_amount: u64) -> Result<()> {
    let clock = Clock::get()?;
    let current_unix_timestamp = clock.unix_timestamp as u64;

    // Payment (USDC) decimals
    let payment_decimal_value =
(10u64).pow(ctx.accounts.payment_mint.decimals as u32);

    // Presale token (Token-2022) decimals
    let token_decimal_value =
(10u64).pow(ctx.accounts.token_mint.decimals as u32);

    require!(
        current_unix_timestamp >=
ctx.accounts.presale_account.start_time &&
            current_unix_timestamp <=
ctx.accounts.presale_account.end_time,
        PresaleErrorCodes::InvalidTime
    );

    // Check token_amount is within the buyable range
    require!(
        token_amount >=
ctx.accounts.presale_account.minimum_buyable_amount &&
            token_amount <=
ctx.accounts.presale_account.maximum_buyable_amount,
        PresaleErrorCodes::InvalidPurchaseAmount
    );

    // Calculate the USDC payment amount
    let payment_amount =
        ctx.accounts.presale_account.token_price_in_usdc *
(token_amount / token_decimal_value);
    msg!("Payment amount: {}", (payment_amount as f64) /
(payment_decimal_value as f64));

    // Ensure payment_amount is greater than 0
    require!(payment_amount > 0, PresaleErrorCodes::ZeroPaymentAmount);

    if payment_amount > 0 {
        // Transfer USDC payment from buyer to creator
        anchor_spl::token::transfer_checked(
            ctx.accounts.transfer_usdc_payment_to_creator(),
            payment_amount,
            ctx.accounts.payment_mint.decimals
        )?;
        msg!("Payment Done");

        // Now transfer the presale tokens (Token-2022) from presale to
buyer
        let bump = ctx.accounts.presale_account.bump;
        anchor_spl::token_interface::transfer_checked(

ctx.accounts.transfer_presale_token_to_buyer().with_signer(&[&[PRESALE_
SEED, &[bump]]]),
            token_amount,
            ctx.accounts.token_mint.decimals
        )?;
        msg!("Token bought successfully");

        ctx.accounts.presale_account.total_tokens -= token_amount;
    } else {
    }

    emit!(PurchaseTokenEvent {
        buyer: ctx.accounts.buyer.key(),
        bought_token_amount: token_amount,
    });
    Ok(())
```

Recommendation

The calculation should be adjusted to ensure accurate handling of decimals in both the payment and token transfer logic. This ensures that users pay the correct amount for the tokens they receive, preserving fairness and the intended functionality of the presale.

RCC - Redundant Conditional Check

Criticality	Minor / Informative
Location	processor/purchase_token.rs#L5
Status	Unresolved

Description

The function `purchase_token_handler` includes a conditional check that is redundant due to a preceding validation requiring the same condition. This makes the subsequent check unnecessary and introduces redundant code. Additionally, the else statement associated with this conditional check is empty, contributing no functional value to the logic.

```
require!(payment_amount > 0, PresaleErrorCodes::ZeroPaymentAmount);

    if payment_amount > 0 {
        // Transfer USDC payment from buyer to creator
        anchor_spl::token::transfer_checked(
            ctx.accounts.transfer_usdc_payment_to_creator(),
            payment_amount,
            ctx.accounts.payment_mint.decimals
        )?;
        msg!("Payment Done");

        // Now transfer the presale tokens (Token-2022) from presale to
buyer
        let bump = ctx.accounts.presale_account.bump;
        anchor_spl::token_interface::transfer_checked(

ctx.accounts.transfer_presale_token_to_buyer().with_signer(&[&[PRESALE_
SEED, &[bump]]]),
            token_amount,
            ctx.accounts.token_mint.decimals
        )?;
        msg!("Token bought successfully");

        ctx.accounts.presale_account.total_tokens -= token_amount;
    } else {
    }
```

Recommendation

The redundant conditional check should be removed to simplify the function and reduce unnecessary complexity. This ensures that the logic remains concise and focuses on essential operations. Removing the empty else block further enhances code clarity and prevents potential confusion

IKCU - Inconsistent Key Comparison Use

Criticality	Minor / Informative
Location	processor/revoke.rs#L26 processor/withdraw_token.rs#L45
Status	Unresolved

Description

The smart contract uses two different approaches for comparing public keys: one with the `as_ref()` method and one without. While both methods are technically correct and achieve the same result, the inconsistency reduces code readability and may cause confusion for developers maintaining the contract. Using a uniform approach throughout the codebase is essential for clarity and simplicity.

```
#[account(
        mut,
        constraint = authority.key().as_ref() ==
presale_program_data.super_authority.key().as_ref()
        @PresaleErrorCodes::Unauthorized,
    )]
    pub authority: Signer<'info>,

#[account(
        mut,
        seeds = [CREATOR_SEED, authority.key().as_ref()],
        bump,
        constraint = creator_account.creator.key() == authority.key()
        @PresaleErrorCodes::InvalidCreator
    )]
    pub creator_account: Box<Account<'info, CreatorAccount>>,
```

Recommendation

It is recommended to adopt a single, consistent method for key comparisons. This change improves readability and ensures that the code follows a clear and unified style, making it easier for future developers to understand and maintain.

PPR - Potential Presale Reinitialization

Criticality	Minor / Informative
Location	processor/create_presale.rs#L20
Status	Unresolved

Description

The current implementation of the presale creation function permits multiple invocations that can effectively reinitialize or overwrite the presale’s configuration. The use of `init_if_needed` on key presale-related accounts and the conditions allowing a new presale start time to be set after a previous one allows the authority to restart the presale. This creates a situation where the presale may begin again, potentially contradicting expectations that once it is started and concluded, it should not be altered in such a fundamental way. Additionally,the presale can end abruptly, if it is currently active, by setting a new start time in the future. Furthermore, the presence of a separate update function, which allows only the extension of the end time while a presale is still ongoing, suggests that the intended design likely aims to restrict the ability to modify critical presale parameters after initiation. The combination of these behaviors creates a lack of clarity around how the presale is meant to operate over its lifecycle, and it may enable scenarios where a presale can be abruptly redefined or relaunched, potentially confusing participants and undermining trust.

```
pub fn create_presale_handler(ctx: Context<CreatePresale>, args:
CreatePresaleArgs) -> Result<()> {
    let presale_account = &mut ctx.accounts.presale_account;

    msg!("Start time: {}", presale_account.start_time);
    require!(
        presale_account.start_time == 0 || presale_account.end_time <=
args.start_time,
        PresaleErrorCodes::PresaleAlreadyActive
    );

    // uncomment at deployment
    let clock = Clock::get()?;
    let current_unix_timestamp = clock.unix_timestamp as u64;
    msg!(
        "currentTime :{} , start Time: {} & endt Time: {} ",
        current_unix_timestamp,
        args.start_time,
        args.end_time
    );
    msg!("Admin Token ATA Address: {}",
ctx.accounts.admin_token_ata.key());
    msg!("Presale Token ATA Address: {}",
ctx.accounts.presale_token_ata.key());
    msg!("creator account Address: {}",
ctx.accounts.creator_account.creator.key());

    require!(
        current_unix_timestamp < args.start_time && args.end_time >
args.start_time,
        PresaleErrorCodes::InvalidTime
    );
    require!(
        args.maximum_buyable_amount > args.minimum_buyable_amount,
        PresaleErrorCodes::InvalidPurchaseAmount
    );
    presale_account.authority = args.authority.key();
    presale_account.start_time = args.start_time;
    presale_account.end_time = args.end_time;
    presale_account.minimum_buyable_amount =
args.minimum_buyable_amount;
    presale_account.maximum_buyable_amount =
args.maximum_buyable_amount;
    presale_account.total_tokens += args.presale_token_amount;
    presale_account.token_price_in_usdc = args.token_price_in_usdc;
    presale_account.bump = ctx.bumps.presale_account;

    anchor_spl::token_interface::transfer_checked(
        ctx.accounts.transfer_token_to_presale_ata(),
        args.presale_token_amount,
        ctx.accounts.token_mint.decimals
    )?;

    emit!(CreatePresaleEvent {
        authority: ctx.accounts.authority.key(),
        token_amount: args.presale_token_amount,
        start_time: args.start_time,
        end_time: args.end_time,
        minimum_buyable_amount: args.minimum_buyable_amount,
        maximum_buyable_amount: args.maximum_buyable_amount,
        token_price_in_usdc: args.token_price_in_usdc,
    });
    Ok(())
}
```

Recommendation

It is recommended to re-evaluate the presale logic to establish a clear and consistent lifecycle. The core values and parameters of the presale should not be modifiable through the creation function after it has already started, unless the explicit business logic allows for a safe and transparent process to do so. By clarifying the intended behavior and ensuring that the creation function cannot overwrite a running or completed presale, the contract's integrity and the participants' expectations will be better preserved.

PCR - Program Centralization Risk

Criticality	Minor / Informative
Location	processor/create_presale.rs#L20 processor/update_presale.rs#L9
Status	Unresolved

Description

The program's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion. Specifically, the program owner has the authority to restart the presale multiple times, even after it has concluded. This means that the owner can modify key parameters, such as the start and end times, and relaunch the presale at will. Additionanlly, the program owner has the authority to update the parameters of the presale by calling the `update_presale` function.

```
pub fn create_presale(ctx: Context<CreatePresale>, args:
CreatePresaleArgs) -> Result<()> {
        create_presale::create_presale_handler(ctx, args)
    }

pub fn update_presale(ctx: Context<UpdatedData>, args:
UpdatePresaleArgs) -> Result<()> {
        update_presale::update_presale_handler(ctx, args)
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

RFN - Redundant Field Names

Criticality	Minor / Informative
Location	processor/update_presale.rs#L29 processor/revoke.rs#L14
Status	Unresolved

Description

There are redundant field names in the struct initialization process. In the files `update_presale.rs` and `revoke.rs`, field names are unnecessarily repeated when initializing the struct with variables of the same name.

```
emit!(UpdatePresaleEvent {
        previous_end_time: previous_end_time,
        end_time: presale_account.end_time,
    });

emit!(RevokeAccessEvent {
        old_authority: old_authority,
    });
```

Recommendation

Avoid specifying the field name explicitly when the variable name matches the field name in the struct. This streamlines the struct initialization process, enhances readability, and aligns with Rust's recommended coding practices.

SSI - Single Seed Initialization

Criticality	Minor / Informative
Location	processor/create_presale.rs#L78
Status	Unresolved

Description

The `presale_account` uses a constant seed in its initialization, which restricts it to managing only one token mint. If multiple token mints are intended to be supported, the current design causes all presales to overwrite the same account due to the `init_if_needed` directive. This creates potential issues with overwriting data and limits the contract’s functionality for handling multiple token mints.

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

    #[account(
        mut,
        seeds = [CREATOR_SEED, authority.key().as_ref()],
        bump,
        constraint = creator_account.creator.key().as_ref() ==
authority.key().as_ref()
        @PresaleErrorCodes::InvalidCreator
    )]
    pub creator_account: Box<Account<'info, CreatorAccount>>,

    #[account(
        init_if_needed,
        payer = authority,
        seeds = [PRESALE_SEED],
        bump,
        space = 8 + PresaleAccount::INIT_SPACE
    )]
    pub presale_account: Box<Account<'info, PresaleAccount>>,

    #[account(
        init_if_needed,
        payer = authority,
        associated_token::mint = token_mint,
        associated_token::authority = presale_account
    )]
    pub presale_token_ata: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(
        mut,
    )]
    pub admin_token_ata: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(seeds = [PRESALE_SEED, PROGRAM_DATA_SEED], bump =
presale_program_data.bump)]
    pub presale_program_data: Box<Account<'info, PresaleProgramData>>,
    #[account(
        mut,
        constraint=token_mint.key().as_ref() ==
presale_program_data.token_mint.as_ref()
        @PresaleErrorCodes::InvalidMintedToken,
    )]
    pub token_mint: Box<InterfaceAccount<'info, Mint>>,
    pub token_program: Program<'info, Token2022>,
    pub associated_token_program: Program<'info, AssociatedToken>,
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,
}
```

Recommendation

It is recommended to clarify the intended functionality of the presale. If only one token mint is supported, remove `init_if_needed` and ensure proper error handling for attempts to initialize multiple presales. If multiple token mints are intended, include the token mint as part of the seed and modify the design of `presale_program_data` to support multiple token mints effectively. This ensures the contract functions as intended and avoids unintended overwrites or limitations.

UAC - Unused Account Context

Criticality	Minor / Informative
Location	processor/purchase_token.rs#L134
Status	Unresolved

Description

The `creator_account` is declared in the transaction context but is not utilized anywhere within the function logic or referenced by other accounts in the context. This indicates either an oversight where the account was intended to serve a purpose but is missing its usage in the logic, or that it is unnecessary and should be removed.

```
#[account(
        mut,
        seeds = [CREATOR_SEED, creator_account.creator.key().as_ref()],
        bump
    )]
    pub creator_account: Box<Account<'info, CreatorAccount>>,
```

Recommendation

Determine the intended purpose of the `creator_account`. If it is essential to the transaction logic, its intended usage should be implemented accordingly. Otherwise, if it serves no functional purpose, it should be removed to avoid confusion and to streamline the context. This ensures clarity, reduces unnecessary complexity, and improves the maintainability of the contract.

UEC - Unused Error Codes

Criticality	Minor / Informative
Location	error.rs#L4
Status	Unresolved

Description

There are error codes that are not used anywhere in the programs. Maintaining unused error codes can introduce confusion and may give the false impression that specific access control mechanisms are implemented when they are not. It also increases the complexity of understanding the contract and poses a risk of errors in future updates if developers mistakenly believe this error code is actively used.

```
#[error_code]
pub enum PresaleErrorCodes {
    ...
    #[msg("Invalid amount")]
    InvalidAmount,
    ...
    #[msg("Error during calculating the price ")]
    CalculationError,
    ...
    #[msg("There is not any presale")]
    PresaleDoesnotExist,
    ...
}
```

Recommendation

Review the contract to determine if there is an intended use for those error codes. If they are required, ensure that they are correctly implemented where needed. Otherwise, if there is no need for those error codes, consider removing them to keep the codebase clean and maintainable. Clear and concise error handling contributes to a more efficient and secure smart contract design.

Summary

The GroWealth contract implements a presale mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope'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 regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document 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 represents an extensive assessment 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 Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope'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. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io