H1: Users Can Withdraw Full Stake Despite Slashing

Description

The Grass protocol’s staking mechanism enables slashing of stake pool tokens, transferring a portion to a designated destination. This is handled by the `slashing_handler` function:

```rust
pub fn slashing_handler<'info>(
    ctx: Context<Slashing>,
    amount: u64,
    router: u8,
    is_locked: u8
) -> Result<()> {
    let stake_pool = &mut ctx.accounts.stake_pool.load_mut()?;
    let pool = &mut stake_pool.reward_pools[usize::from(router)];
    pool.is_locked = is_locked;

    let cpi_ctx = CpiContext {
        program: ctx.accounts.token_program.to_account_info(),
        accounts: Transfer {
            from: ctx.accounts.vault.to_account_info(),
            to: ctx.accounts.vault.to_account_info(),
            authority: ctx.accounts.stake_pool.to_account_info(),
        },
        remaining_accounts: Vec::new(),
        signer_seeds: &[stake_pool_signer_seeds!(stake_pool)],
    };
    token::transfer(cpi_ctx, amount)?;
    Ok(())
}
```

The function intends to deduct tokens proportionally from users staking with a delegate. However, the `withdraw_handler` function does not adjust for slashed amounts, allowing users to withdraw their entire original deposit:

```
pub fn withdraw_handler<'info>(ctx: Context<'_, '_, 'info, 'info, Withdraw<'info>>) -> Result<()> {
    ctx.accounts.validate_stake_pool_and_owner()?;
    ctx.accounts.claim_base.stake_deposit_receipt.validate_unlocked()?;
    {
        let mut stake_pool = ctx.accounts.claim_base.stake_pool.load_mut()?;
        let total_staked = stake_pool
            .total_weighted_stake_u128()
            .checked_sub(
                ctx.accounts.claim_base.stake_deposit_receipt.effective_stake_u128(),
            )
            .unwrap();
        stake_pool.total_weighted_stake = u128(total_staked.to_le_bytes());
    }
    ctx.accounts.transfer_staked_tokens_to_owner()?;
    ctx.accounts.close_stake_deposit_receipt()?;
    Ok(())
}
```

This flaw allows users to bypass slashing penalties, claiming their full stake regardless of deductions.

Impact

This vulnerability undermines the staking pool’s fairness, enabling users to withdraw unpenalized amounts, potentially depleting the pool. Later withdrawers may face shortages, disrupting the protocol’s slashing enforcement.

Recommendation

Revise the `withdraw_handler` function to account for slashed tokens, ensuring withdrawals reflect proportional deductions based on pool penalties.

H2: Unrestricted Access to TokenAirdrop Critical Functions

Description

The `TokenAirdrop` program includes `set_admin` and `lock_claims` functions for critical operations: updating the admin and toggling airdrop claim status. These should be restricted to an admin account, but the functions lack caller verification:

```rust
#[allow(clippy::result_large_err)]
pub fn set_admin(ctx: Context<SetAdmin>) -> Result<()> {
    handle_set_admin(ctx)
}

#[allow(clippy::result_large_err)]
pub fn lock_claims(ctx: Context<LockClaims>, is_locked: bool) -> Result<()> {
    handle_lock_claims(ctx, is_locked)
}
```

This oversight allows any user to invoke these functions, enabling unauthorized changes to the admin or claim lock status.

Impact

Lack of access controls risks unauthorized admin changes or claim disruptions, potentially leading to mismanagement, fund loss, or denial of service for legitimate airdrop participants.

Recommendation

Add caller verification to `set_admin` and `lock_claims`, ensuring only the admin account can execute these sensitive operations.

H3: Insufficient Constraints in ClaimBase Structure

Description

The `ClaimBase` structure, used in staking operations, includes `stake_pool` and `stake_deposit_receipt` accounts, which should be Program Derived Addresses (PDAs) owned by the Staking program. However, the structure lacks ownership validation:

```rust
#[derive(Accounts)]
pub struct ClaimBase<'info> {
    /// Owner of the StakeDepositReceipt
    #[account(mut)]
    pub owner: Signer<'info>,

    // StakePool the StakeDepositReceipt belongs to
    #[account(mut)]
    pub stake_pool: AccountLoader<'info, StakePool>,

    /// StakeDepositReceipt of the owner that will be used to claim respective rewards
    #[account(
      mut,
      has_one = owner @ ErrorCode::InvalidOwner,
      has_one = stake_pool @ ErrorCode::InvalidStakePool,
    )]
    pub stake_deposit_receipt: Account<'info, StakeDepositReceipt>,
}
```

The constraints verify only that `stake_deposit_receipt` references the `owner` and `stake_pool`, without ensuring these accounts are owned by the Staking program, allowing attackers to use crafted accounts.

Impact

This vulnerability enables attackers to supply malicious `stake_pool` and `stake_deposit_receipt` accounts, manipulating the claim process to siphon funds, which could lead to significant financial losses for the protocol.

Recommendation

Add ownership checks to the `ClaimBase` structure, ensuring `stake_pool` and `stake_deposit_receipt` are owned by the Staking program, using Anchor's ownership validation `ID` to enforce correct account types and discriminators.

H4: Missing Authority Check in Slashing Function

Description

The `slashing_handler` function, responsible for slashing stake pool tokens, defines an `authority` field as a signer in the `Slashing` struct but fails to verify that this signer matches the authorized stake pool authority:

```rust
#[derive(Accounts)]
pub struct Slashing<'info> {
    // ...
    #[account(
        mut,
        has_one = vault @ ErrorCode::InvalidStakePoolVault,
        has_one = stake_mint @ ErrorCode::InvalidAuthority,
    )]
    pub stake_pool: AccountLoader<'info, StakePool>,
    // ...
}
```

This omission allows any user with a valid signer account to invoke `slashing_handler` and perform unauthorized slashing operations on the stake pool.

Impact

Unauthorized slashing could lead to improper token deductions from the stake pool, causing financial losses for stakers and disrupting the protocol's penalty mechanism, potentially undermining trust and functionality.

Recommendation

Add a constraint to the `Slashing` struct to verify that the signer matches the stake pool's authority, such as `has_one = authority @ ErrorCode::InvalidAuthority`, and ensure the `ErrorCode` enum includes an `InvalidAuthority` variant to reject unauthorized callers.

H5: Rewards Claimable Despite Locked Reward Pool

Description

The Grass protocol's `slashing_handler` function sets an `is_locked` flag on a reward pool to restrict actions, such as reward claims:

```rust
pub fn slashing_handler<'info>(
    ctx: Context<Slashing>,
    amount: u64,
    router: u8,
    is_locked: u8
) -> Result<()> {
    let stake_pool = &mut ctx.accounts.stake_pool.load_mut()?;
    let pool = &mut stake_pool.reward_pools[usize::from(router)];
    pool.is_locked = is_locked;

    let cpi_ctx = CpiContext {
        program: ctx.accounts.token_program.to_account_info(),
        accounts: Transfer {
            from: ctx.accounts.vault.to_account_info(),
            to: ctx.accounts.vault.to_account_info(),
            authority: ctx.accounts.stake_pool.to_account_info(),
        },
        remaining_accounts: Vec::new(),
        signer_seeds: &[stake_pool_signer_seeds!(stake_pool)],
    };
    token::transfer(cpi_ctx, amount)?;
    Ok(())
}
```

However, the `transfer_all_claimable_rewards` function in `ClaimBase` does not check the `is_locked` flag, allowing rewards to be claimed from a locked pool:

```
pub fn transfer_all_claimable_rewards(
    &self,
    remaining_accounts: &[AccountInfo<'info>],
) -> Result<[u64; MAX_REWARD_POOLS]> {
    for (index, reward_pool) in stake_pool.reward_pools.iter().enumerate() {
        if reward_pool.is_empty() {
            continue;
        }
        // ... (reward transfer logic)
    }
}
```

This bypasses the intended lock mechanism, permitting unauthorized reward claims.

Impact

This vulnerability allows users to claim rewards from locked pools, undermining the protocol's control mechanisms and potentially leading to unauthorized fund withdrawals, which could destabilize the reward distribution system.

Recommendation

Modify the `transfer_all_claimable_rewards` and `update_reward_pools_last_amount` functions to skip locked reward pools (e.g., check `reward_pool.is_locked != 0`) and add error handling to notify users when claims are blocked due to a locked pool.

M1: StakeWeightTokens Not Burned on Unstaking

Description

The Grass protocol mints `StakeWeightTokens` to users upon staking, intended to represent their stake for withdrawal purposes, as implemented in the `mint_staked_token_to_user` function:

```rust
pub fn mint_staked_token_to_user(&self, effective_amount: u64) -> Result<()> {
    let stake_pool = self.stake_pool.load()?;
    let signer_seeds: &[&[&[u8]]] = &[stake_pool_signer_seeds!(stake_pool)];
    let cpi_ctx = CpiContext::new_with_signer(
        self.token_program.to_account_info(),
        MintTo {
            mint: self.stake_mint.to_account_info(),
            to: self.destination.to_account_info(),
            authority: self.stake_pool.to_account_info(),
        },
        signer_seeds,
    );
    token::mint_to(cpi_ctx, effective_amount)
}
```

However, the withdrawal function does not burn these tokens, as the burning logic is commented out:

```rust
//ctx.accounts.burn_stake_weight_tokens_from_owner()?;
```

This allows users to retain `StakeWeightTokens` after unstaking, violating the intended token lifecycle.

Impact

Retaining `StakeWeightTokens` after unstaking could enable users to misuse these tokens, potentially disrupting the protocol's staking accounting or allowing unauthorized claims, which may lead to economic inconsistencies or exploits.

Recommendation

Reinstate the burning logic in the withdrawal function (e.g., uncomment `burn_stake_weight_tokens_from_owner`) to ensure `StakeWeightTokens` are burned upon unstaking, maintaining the protocol's token integrity.

M2: Attacker Can Block Deposits with Minimal or Zero Amount

Description

The Grass protocol's deposit instruction creates a `StakeDepositReceipt` account, a PDA derived from a user-specified `nonce`, `owner`, `stake_pool`, and a string. The `owner` is not required to be a signer:

```
#[account(
  init,
  seeds = [
    &nonce.to_le_bytes(),
    owner.key().as_ref(),
    stake_pool.key().as_ref(),
    b"stakeDepositReceipt",
  ],
  bump,
  payer = payer,
  space = 8 + StakeDepositReceipt::LEN,
)]
pub stake_deposit_receipt: Account<'info, StakeDepositReceipt>,
```

This allows an attacker to front-run a user's deposit transaction by creating a `StakeDepositReceipt` with the same `nonce` and `owner` but a zero or minimal amount, causing the legitimate transaction to fail due to an existing PDA.

Impact

This vulnerability enables attackers to censor user deposits by front-running with low-cost transactions, disrupting user participation and potentially deterring engagement with the staking protocol.

Recommendation

Require the `owner` to be a signer of the deposit transaction or replace the `nonce` with an incremental counter or random value to prevent attackers from preemptively initializing the `StakeDepositReceipt` PDA.