Pashov Audit Group

BTR Security Review

Conducted by:
ctrus
0xeix
FrankCastle
ZeroTrust01

June 29th 2025 - July 5th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

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

3. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

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

4. About BTR

BTR is a BTC-backed reserve on Solana, governed by staking, voting, and vault-based investment products, with a treasury-backed supply of 10 million $BTR. The audit scope covers the security and functionality of BTR's smart contracts including staking-voting logic and vault maturity mechanisms, while excluding off-chain components like the ATM engine.

5. Executive Summary

A time-boxed security review of the **btrfi/btr-contracts** repository was done by Pashov Audit Group, during which **ctrus**, **0xeix**, **FrankCastle**, **ZeroTrust01** engaged to review **BTR**. A total of **25** issues were uncovered.

Protocol Summary	
Project Name	BTR
Protocol Type	Treasury Vault
Timeline	June 29th 2025 - July 5th 2025

Review commit hash:
- 38a176f5ed83f7ca346c81401519346739129e7e (btrfi/btr-contracts)

Fixes review commit hash:
- 6e4ae85ba1930afc2e39de31a5f608bd22043f83 (btrfi/btr-contracts)

Scope

`admin_functions.rs` `approve_update_signer.rs` `approve_voting.rs` `claim.rs` `create_staker_info.rs` `create_update_signer.rs` `create_voter_info.rs` `create_voting.rs` `initialize_global_data.rs` `mod.rs` `stake_btr.rs` `unstake_btr.rs` `vote.rs` `global_data.rs` `mint_event.rs` `staker_info.rs` `update_signer_event.rs` `voter_info.rs` `voting_event.rs` `error.rs` `lib.rs` `approve_request.rs` `cancel_request.rs` `create_user_entry.rs` `create_vault_request.rs` `initialize.rs` `stake.rs` `unlock_vault.rs` `unstake.rs` `user_entry.rs` `vault_pool.rs` `get_price.rs`

6. Findings

Findings count

Severity	Amount
Critical	5
High	2
Medium	5
Low	13
Total findings	**25**

Summary of findings

ID	Title	Severity	Status
[C-01]	Funds permanently locked in treasury account with no withdrawal mechanism	Critical	Resolved
[C-02]	Only one voter is allowed per voting event due to incorrect PDA seeds	Critical	Resolved
[C-03]	Missing user authorization in `GovBTR` token claim function	Critical	Resolved
[C-04]	`voter_info` account does not include voter address in seeds	Critical	Resolved
[C-05]	Missing access control in `claim` function allows unauthorized token claims	Critical	Resolved
[H-01]	Users cannot claim GOVBTR tokens when voting event status is AllMinted	High	Resolved
[H-02]	Voting events limited to 255 maximum due to type mismatch	High	Resolved
[M-01]	Staking and vault accounts not properly closed locking rent-exempt funds	Medium	Resolved
[M-02]	Staking allows surpassing `max_usdc_capacity` causing reward mismatch	Medium	Acknowledged
[M-03]	High likelihood of overflow in reward calculation due to `f64` precision limit	Medium	Resolved

ID	Title	Severity	Status
[M-04]	Denial of service via associated token account pre-creation	Medium	Resolved
[M-05]	Users cannot unstake after 7 days if voting event duration is longer	Medium	Resolved
[L-01]	Mint and gov authorities not properly managed for rent exemption	Low	Resolved
[L-02]	`VaultPool` bump not stored and redundant assignment in `create_vault_request`	Low	Resolved
[L-03]	Incorrect assignment of `total_minted` in vote function	Low	Resolved
[L-04]	Vault creation blocked if one signer is compromised	Low	Resolved
[L-05]	Variable naming does not reflect its purpose	Low	Resolved
[L-06]	Staking duration can be invalid due to prematurely set `staking_end_time`	Low	Resolved
[L-07]	Improper signer validation leads to permanent denial of service	Low	Resolved
[L-08]	Inconsistent TWAP staleness check allows use of stale price data	Low	Resolved
[L-09]	`VaultPool` and escrow token rent lamports not refunded on cancellation	Low	Resolved
[L-10]	`UpdateSignersEvent` rent refund goes to approver not creator	Low	Acknowledged
[L-11]	`MintEvent` rent refund goes to approver not creator	Low	Resolved
[L-12]	Missing recipient validation for voting event and associated Mint event	Low	Resolved
[L-13]	Missing seed validation allows unauthorized token approval	Low	Resolved

Critical findings

[C-01] Funds permanently locked in treasury account with no withdrawal mechanism

Severity

Impact: High

Likelihood: High

Description

In `stake.rs` when a vault reaches its maximum capacity (`vault.total_staked >= vault.max_usdc_capacity`), the contract transfers the staked tokens from the escrow account to the treasury account:

```
let cpi_accounts = Transfer {
    from: ctx.accounts.escrow_token_account.to_account_info(),
    to: ctx.accounts.treasury_account.to_account_info(),
    authority: vault_pool_info,
};
let cpi_ctx = CpiContext::new_with_signer(
    ctx.accounts.token_program.to_account_info(),
    cpi_accounts,
    signer_seeds,
);
token::transfer(cpi_ctx, vault.total_staked)?;
```

The issue is that there is no functionality implemented in the contract to withdraw these funds from the treasury account once they are transferred. This effectively locks the funds permanently in the treasury account with no recovery mechanism.

This creates a significant risk of fund loss whenever a vault reaches its maximum capacity, which is an expected operation in the protocol.

Recommendations

Implement a secure withdrawal mechanism for the treasury account, allowing authorized entities (like admins or governance) to access these funds when needed.

[C-02] Only one voter is allowed per voting event due to incorrect PDA seeds

Severity

Impact: High

Likelihood: High

Description

The `CreateVoterInfo` struct uses a PDA (Program Derived Address) that only includes the voting event's public key as a seed, without incorporating the voter's public key:

```
#[account(
    init,
    payer = voter,
    space = 8 + 32 + 8 + 32, // Account discriminator + Pubkey + i64 + Pubkey
    seeds = [b"voter_info", voting_event.key().as_ref()],
    bump
)]
pub voter_info: Account<'info, VoterInfo>,
```

This design severely limits the protocol's functionality by allowing only one `VoterInfo` account per voting event. When a second user attempts to call `create_voter_info` for the same voting event, the transaction will fail because the account already exists.

The impacts of this issue include:

1. **Governance functionality broken**: Only one user can participate in each voting event, defeating the purpose of decentralized governance.
2. **Contradicts design intentions**: The `VotingEvent` struct includes a `min_participants` field and a `current_participants` counter, clearly indicating the system was designed for multiple voters per event.

Recommendations

Modify the `VoterInfo` PDA derivation to include both the voting event key and the voter's public key:

```
#[account(
    init,
    payer = voter,
    space = 8 + 32 + 8 + 32, // Account discriminator + Pubkey + i64 + Pubkey
    seeds = [b"voter_info", voting_event.key().as_ref(), voter.key().as_ref()],
    bump
)]
pub voter_info: Account<'info, VoterInfo>,
```

This change should be applied consistently across the codebase, including in: - `staking/programs/staking/src/instructions/create_voter_info.rs`. - `staking/programs/staking/src/instructions/vote.rs`. - `staking/programs/staking/src/instructions/claim.rs`.

[C-03] Missing user authorization in `GovBTR` token claim function

Severity

Impact: High

Likelihood: High

Description

The `ClaimGovBtr` struct in the claim function has faulty account constraints for `voter_info`. It allows any user to claim GovBTR tokens from any valid `voter_info` account, regardless of whether those tokens actually belong to them.

The current implementation only validates that the `voter_info` account is a PDA derived from "voter_info" and the voting event's key, but doesn't verify that this account belongs to the transaction signer:

```
#[account(
    mut,
    seeds = [b"voter_info", voting_event.key().as_ref()],
    bump
)]
pub voter_info: Account<'info, VoterInfo>,
```

As a result, a malicious user can: 1. Find valid `voter_info` accounts associated with any voting event. 2. Pass these accounts to the claim instructions. 3. Successfully transfer tokens from the escrow to their own account. 4. Effectively steal tokens that rightfully belong to other users.

This vulnerability could lead to significant financial losses for legitimate voters.

Recommendations

Add a constraint to verify that the `voter_info` account belongs to the transaction signer:

```
#[account(
    mut,
    seeds = [b"voter_info", voting_event.key().as_ref(), user.key().as_ref()],
    bump,
    constraint = voter_info.voter == user.key() // Assuming VoterInfo has an owner field
)]
pub voter_info: Account<'info, VoterInfo>,
```

This change ensures that users can only claim GovBTR tokens from voter accounts they actually own.

[C-04] `voter_info` account does not include voter address in seeds

Severity

Impact: High

Likelihood: High

Description

In the current implementation of the `CreateVoterInfo` instruction, the `voter_info` account is derived using the following seed:

```
seeds = [b"voter_info", voting_event.key().as_ref()],
```

This means only **one** `voter_info` account can exist per `voting_event`, regardless of how many users participate. Since the PDA does not include the `voter`'s public key, **any attempt by a second user to create a** `voter_info` **account will fail** due to a PDA collision.

This results in:

- **Inability for multiple users to vote or stake** `govBTR` in a voting event.
- **Protocol-wide loss of value**, as users are unable to lock tokens and gain rewards or voting influence.
- **Complete freeze of the voting mechanism**, effectively breaking governance participation.

Here is the current code excerpt:

```
#[account(
    init,
    payer = voter,
    space = 8 + 32 + 8 + 32,
    seeds = [b"voter_info", voting_event.key().as_ref()],
    bump
)]
pub voter_info: Account<'info, VoterInfo>,
```

Recommendations

Modify the PDA derivation to include the `voter`'s public key in the seed so that **each voter can have their own** `voter_info` **account per voting event**:

```
#[account(
    init,
    payer = voter,
    space = 8 + 32 + 8 + 32,
    seeds = [b"voter_info", voting_event.key().as_ref(), voter.key().as_ref()],
    bump
)]
pub voter_info: Account<'info, VoterInfo>,
```

This ensures uniqueness per `(voting_event, voter)` pair and enables proper tracking of each participant's stake and voting behavior.

Also, make sure to update all instructions that derive or access `voter_info` PDAs to match the new seed structure.

[C-05] Missing access control in `claim` function allows unauthorized token claims

Severity

Impact: High

Likelihood: High

Description

The `claim` function lacks proper access control, allowing **any signer** to claim `govBTR` tokens stored in a `voter_info` account—even if they are not the original voter who owns that account. This oversight introduces several critical vulnerabilities:

1. **Unauthorized Token Transfers:** The function transfers all `govBTR` tokens from the escrow to the `user_govbtr_account`, but it does **not** verify that the `user` is the same as `voter_info.voter`.
2. **Voting Manipulation:** An attacker can loop through various `voter_info` accounts and claim tokens for themselves, effectively **aggregating infinite voting power**. This undermines the governance model and compromises the integrity of the system.
3. **Fund Loss for Legitimate Voters:** Legitimate voters will lose both their tokens and their right to vote, as someone else can drain their tokens without their signature or consent.
4. **Rent Theft:** Once the voter's token account is drained, the attacker can also reclaim rent from the now-empty account.

The following code demonstrates the issue:

```
pub struct ClaimGovBtr<'info> {
    #[account(mut)]
    pub user: Signer<'info>,
    ...
}
```

No validation exists to ensure that `user == voter_info.voter`.

Yet tokens are transferred to `user_govbtr_account` as follows:

```
#[account(
    mut,
    token::mint = govbtr_token,
    token::authority = user
)]
pub user_govbtr_account: Account<'info, TokenAccount>,
```

And finally, the actual transfer logic:

```
token::transfer(
    CpiContext::new_with_signer(
        ctx.accounts.token_program.to_account_info(),
        token::Transfer {
            from: ctx.accounts.govbtr_account_escrow.to_account_info(),
            to: ctx.accounts.user_govbtr_account.to_account_info(),
            authority: voting_event.to_account_info(),
        },
        signer_seeds
    ),
    amount,
)?;
```

Recommendations

Add a strict access control check to ensure that only the original voter can claim the tokens. Specifically, assert that the `user` signer matches the `voter` field in the `voter_info` account:

```
require!(
    ctx.accounts.user.key() == ctx.accounts.voter_info.voter,
    CustomError::UnauthorizedClaim
);
```

This check ensures that only the rightful owner of the `voter_info` account can receive the tokens and prevents unauthorized users from draining voting power or funds.

High findings

[H-01] Users cannot claim GOVBTR tokens when voting event status is AllMinted

Severity

Impact: High

Likelihood: Medium

Description

The `claim` function in `staking/programs/staking/src/instructions/claim.rs` only allows users to claim their GOVBTR tokens if a voting event is in either the `Completed` or `Expired` status:

```
require!(voting_event.status == VotingStatus::Completed || voting_event.status == VotingStatus::Expired, CustomError::VotingEventNotCompleted);
```

However, this check does not include the `AllMinted` status, which is a legitimate terminal state for a voting event. A voting event transitions to `AllMinted` when all allocated BTR tokens have been minted:

```
// In approve_mint_event function
if voting_event.total_minted == voting_event.max_btr_mintable {
    voting_event.status = VotingStatus::AllMinted;
}

// In vote function
if voting_event.initial_mint == voting_event.max_btr_mintable {
    voting_event.status = VotingStatus::AllMinted;
}
```

This creates a scenario where users who have voted in an event that transitions to `AllMinted` cannot claim their GOVBTR tokens back, effectively locking their tokens indefinitely. This is problematic because:

1. A voting event can directly transition from `Live` to `AllMinted` without going through the `Completed` status first.
2. Once in `AllMinted`, the voting event has fulfilled its purpose (all tokens minted), and users should be able to reclaim their governance tokens.

Recommendations

Modify the condition in the `claim` function to also allow claims when a voting event is in the `AllMinted` status:

```
require!(
    voting_event.status == VotingStatus::Completed ||
    voting_event.status == VotingStatus::Expired ||
    voting_event.status == VotingStatus::AllMinted,
    CustomError::VotingEventNotCompleted
);
```

This change ensures that users can always claim their GOVBTR tokens once a voting event has reached any terminal state (`Completed` , `Expired` , or `AllMinted`), preventing token lockups.

[H-02] Voting events limited to 255 maximum due to type mismatch

Severity

Impact: High

Likelihood: High

Description

The voting system is designed to support a large number of voting events as evidenced by the use of `u64` for the `id` field in the `VotingEvent` struct. However, there's a critical type mismatch in the codebase:

1. In `staking/programs/staking/src/state/global_data.rs` , the counter for voting events is defined as:

```
pub id: u8,  // Limited to 255 values (0-255)
```

1. While in `staking/programs/staking/src/state/voting_event.rs` , the ID for each voting event is defined as:

```
pub id: u64,  // Supports billions of values
```

1. In `staking/programs/staking/src/instructions/create_voting.rs` , when creating a new voting event, the ID is assigned from the global counter:

```
voting_event.id = global_data.id as u64;  // Cast from u8 to u64
global_data.id = global_data.id + 1;  // Increment counter
```

1. Furthermore, all instructions that reference voting events by ID use `u8` parameters:

```
#[instruction(voting_id: u8)]
```

This type mismatch means that despite the system being designed to support many voting events (as evidenced by `u64` for `VotingEvent.id`), it's actually limited to a maximum of 255 total voting events. For a governance system intended for long-term use, this limitation presents a serious constraint that contradicts the apparent design intent of supporting many

more voting events. Those two accounts are not being closed, so if we are again gonna initialize the `voting_event` and `govbtr_account_escrow` at that address(at which an account already exists) it would create dos.

Recommendations

1. Modify the `GlobalData` struct to use a consistent data type for the ID counter:

```
// In global_data.rs
pub struct GlobalData {
    pub id: u64,  // Change from u8 to u64
    // ...other fields
}
```

1. Update all instruction definitions to consistently use `u64` for voting IDs:

```
#[instruction(voting_id: u64)]
```

This change will ensure that the system can support the intended number of voting events without arbitrary limitations from data type choices.

Medium findings

[M-01] Staking and vault accounts not properly closed locking rent-exempt funds

Severity

Impact: Medium

Likelihood: Medium

Description

Across several parts of the program, important on-chain accounts are not closed when they become obsolete, resulting in permanent locking of users' rent-exempt lamports and unnecessary state bloat. Specifically:

- In the `unstake()` function (vault staking), when `user_entry.total_staked == 0`, the `user_entry` account is no longer usable for claims. However, it remains open, and the rent stays locked.
- In the `unstake_btr()` function (BTR staking), when `staker_info.total_staked == 0`, the `staker_info` account similarly becomes useless but is not closed.
- In the `CancelRequest` flow (vault creation), when a vault request is canceled, the associated `vault_pool` account becomes defunct, yet it is not closed to reclaim the rent-exempt funds.

Leaving these accounts open unnecessarily consumes SOL for rent and increases the on-chain account footprint, which may have long-term cost implications for both users and the program.

Recommendations

Implement proper closure of these accounts once they become obsolete:

- For staking accounts (`user_entry` , `staker_info`), close them when `total_staked` drops to zero. Use Anchor's `#[account(close = user)]` attribute or manual lamport transfer and deallocation logic.
- For vault pool accounts (`vault_pool`), close them upon cancellation of the vault request. For example, update the `CancelRequest` context to:

```
#[account(
    mut,
    seeds = [
```

```
        VAULT_POOL_PREFIX.as_bytes(),
        &_vault_id.to_le_bytes()
    ],
    bump,
    close = user
)]
pub vault_pool: Box<Account<'info, VaultPool>>,
```

[M-02] Staking allows surpassing `max_usdc_capacity` causing reward mismatch

Severity

Impact: Medium

Likelihood: Medium

Description

The current implementation **does not enforce** the `max_usdc_capacity` constraint during staking. As a result, users can deposit more USDC than the intended cap, but **only up to** `max_usdc_capacity` **is considered** when calculating `total_reward`. This leads to a mismatch between the actual value staked and the rewards distributed.

```
    if vault.total_staked >= vault.max_usdc_capacity {
          let seeds = &[
            VAULT_POOL_PREFIX.as_bytes(),
            &vault_id.to_le_bytes(),
            &[vault_pool_bump],
        ];
        let signer_seeds = &[&seeds[..]];
        let cpi_accounts = Transfer {
            from: ctx.accounts.escrow_token_account.to_account_info(),
            to: ctx.accounts.treasury_account.to_account_info(),
            authority: vault_pool_info,
        };
        let cpi_ctx = CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            cpi_accounts,
            signer_seeds,
        );
        token::transfer(cpi_ctx, vault.total_staked)?;
        vault.status = VaultStatus::Locked;
        let current_time = Clock::get()?.unix_timestamp as u64;
        vault.lock_end_time = current_time + vault.lock_duration;
    }
    Ok(())
```

Example Scenario

- `max_usdc_capacity = 400`.
- `vault.total_staked = 200`.
- `userA` stakes `800` USDC.

The total staked becomes `1000`, but the reward is still calculated as:

```
total_reward = ((vault.max_usdc_capacity * 10_u64.pow(9)) as f64 / btr_price) as u64;
```

Assuming `btr_price = 1`, the total reward is only `400`, meaning **600 USDC worth of deposits are ignored** in the reward distribution. This leads to:

- **Loss of value** for stakers.
- **Unfair reward allocation**.
- **Broken staking logic**, as the system allows excess deposits but doesn't reflect them in calculations.

Recommendations

There are three potential solutions to fix this logic flaw:

1. **Strict Cap Enforcement**: Revert the staking transaction if the new stake exceed `max_usdc_capacity`.

   ```
   rust require!( vault.total_staked + amount <= vault.max_usdc_capacity, CustomError::StakeExceedsMaxCapacity );
   ```

1. **Auto-adjust Stake Amount**: Allow partial staking up to the remaining capacity:

   ```
   rust let remaining_capacity = vault.max_usdc_capacity - vault.total_staked; let stake_amount = amount.min(remaining_capacity);
   ```

1. **Use Actual Staked Amount in Reward Calculation**: Replace `vault.max_usdc_capacity` with `vault.total_staked` in the reward formula to accurately reflect actual deposits:

   ```
   rust total_reward = ((vault.total_staked as u128 * 10u128.pow(9)) / btr_price as u128) as u64;
   ```

[M-03] High likelihood of overflow in reward calculation due to `f64` precision limit

Severity

Impact: High

Likelihood: Low

Description

There is a high risk of overflow in the reward calculation logic due to the use of `u64`, which has limited precision (approximately 15–17 decimal digits). The relevant code multiplies `vault.max_usdc_capacity` by `10^9`, potentially exceeding `u64`'s precision bounds and causing incorrect rounding or overflow:

```
total_reward = ((vault.max_usdc_capacity * 10_u64.pow(9)) as f64 / btr_price) as u64;
```

Let's break down the overflow scenario:

- `u64` supports up to **19 significant digits**.
- `10^9` is used as a scaling factor to handle fixed-point math.
- If `vault.max_usdc_capacity` exceeds `10^10`, the product exceeds `10^19`, hitting the edge of what `u64` can represent precisely.
- `Max USDC can be used: 18446_744_073` which is 18500 usdc
- This means **any vault configured with more than** `18500 USDC` **will cause an overflow**, which is highly likely and renders this calculation impractical for real-world use.

Impact

- **Permanent Denial of Service**: If this value overflows or panics during execution, the staking or reward distribution logic will become unusable, freezing the system.

Recommendations

Avoid using `f64` for the multiplication of fixed-point financial computations. Instead, perform the multiplication using **wider integer types** (e.g., `u128`) to maintain precision and safety.

Recommended fix:

```
let scaled_capacity = vault.max_usdc_capacity as u128 * 10u128.pow(9);
let total_reward = (scaled_capacity / btr_price as f64) as u64;
```

[M-04] Denial of service via associated token account pre-creation

Severity

Impact: Medium

Likelihood: Medium

Description

In the `CreateStakerInfo` instruction, the program attempts to initialize a new Associated Token Account (ATA) for the governance token:

```
#[account(
    init,
    payer = staker,
    associated_token::mint = gov_token,
    associated_token::authority = staker,
)]
pub gov_token_account: Account<'info, TokenAccount>,
```

This implementation is vulnerable to a denial-of-service attack. An attacker can pre-create the ATA for potential users before they try to become stakers. When a legitimate user tries to call the `create_staker_info` instruction, it will fail because the `init` constraint attempts to initialize an already initialized account, causing the transaction to revert.

Since creating a staker info account is a prerequisite for participating in the staking system, affected users will be completely prevented from staking. This represents a critical denial of service vulnerability that could block legitimate users from accessing core protocol functionality.

Recommendations

Replace the `init` constraint with `init_if_needed` to handle cases where the ATA already exists:

```
#[account(
    init_if_needed,
    payer = staker,
    associated_token::mint = gov_token,
    associated_token::authority = staker,
)]
pub gov_token_account: Account<'info, TokenAccount>,
```

This change ensures that if the ATA already exists, the instruction will still succeed, allowing users to create their staker info regardless of whether their ATA was pre-created by a third party.

[M-05] Users cannot unstake after 7 days if voting event duration is longer

Severity

Impact: Medium

Likelihood: Medium

Description

In the previous audit report, there was a similar finding that states points out this part of the documentation:

```
After the 7-day staking period, users can
unstake their $BTR, regardless of whether they voted.
```

However, if the users have already voted, their GOVBTR has been transferred to the escrow and they can only be redeemed if the voting event is completed:

```
 require!(voting_event.status == VotingStatus::Completed || voting_event.status ==
VotingStatus::Expired, CustomError::VotingEventNotCompleted);
```

So to eventually unstake the BTR tokens, they need to 1) claim their GOVBTR tokens from the escrow (possible only if the event is completed or expired); 2) burn these GOVBTR tokens and get their BTR tokens back. Currently, they can unstake after 7 days:

```
 require!(staker_info.last_staked_time + 7 * 24 * 3600 < clock.unix_timestamp,
CustomError::NotEnoughTime);
```

But it won't be possible to do that if the voting event duration is > 7 days or if it's not yet completed.

Recommendations

Either restrict the duration of the voting event to 7 days or introduce some custom fix changing the current implementation.

Low findings

[L-01] Mint and gov authorities not properly managed for rent exemption

In the `GlobalData` struct, only `mint_authority` is stored but `gov_authority` is not stored.

```
pub struct GlobalData {
    pub id: u8,
    pub authorized_signers: [Pubkey; 5],
    pub min_num_signers: u8,
    pub owner: Pubkey,
    pub btr_token: Pubkey,
    pub govbtr_token: Pubkey,
    pub mint_authority: Pubkey,
    // @audit gov authority should be stored here
    pub request_id: u8,
    pub update_request_id: u8,
    pub bump: u8, // PDA bump for program authority
}
```

However, there is no mechanism to ensure that these authority accounts (mint authority and governance authority) are rent-exempt or even funded at all. These can be garbage collected, leading to a potential loss of authority.

Recommendations

- Explicitly store both `mint_authority` and `gov_authority` in the `GlobalData` struct.
- Ensure that both authority accounts are created as rent-exempt accounts and are properly funded during protocol initialization or later on with some functionality for it.
- Add logic to periodically check and, if necessary, top up these accounts to maintain rent exemption and prevent garbage collection by the network.
- Also we are passing these two accounts separately in many instructions, it would be better if we use these accounts from `GlobalData`.

[L-02] `VaultPool` bump not stored and redundant assignment in `create_vault_request`

In the `create_vault_request` instruction, the bump value for the `vault_pool` PDA is not stored in the `VaultPool` struct. Instead, other instructions (such as `unstake`) must recalculate the bump using `Pubkey::find_program_address` every time it is needed:

```
//@audit should use stored bump
let (_vault_pool_key, vault_pool_bump) = Pubkey::find_program_address(
    &[
        VAULT_POOL_PREFIX.as_bytes(),
```

```
        &vault_id.to_le_bytes(),
    ],
    ctx.program_id,
);
```

This is inefficient. The bump is available at creation time via `ctx.bumps.vault_pool` and should be stored in the account struct for future use.

Additionally, there is a redundant assignment to `lock_end_time`:

```
vault_pool.lock_end_time = 0;
vault_pool.staking_end_time = staking_end_time;
vault_pool.lock_end_time = 0; // assigned twice
```

This is unnecessary and may cause confusion.

Recommendations

- Store the bump value in the `VaultPool` struct at creation time using `ctx.bumps.vault_pool`.
- Refactor all instructions to use the stored bump instead of recalculating it.
- Remove redundant assignments to `lock_end_time` to improve code clarity and maintainability.

[L-03] Incorrect assignment of `total_minted` in vote function

In the `vote` function in `staking/programs/staking/src/instructions/vote.rs`, when a voting event reaches the minimum number of participants, the function incorrectly sets `voting_event.total_minted = voting_event.initial_mint` instead of adding the `initial_mint` amount to the existing total.

```
mint_tokens(
    ctx.accounts.token_program.to_account_info(),
    ctx.accounts.btr_token.to_account_info(),
    ctx.accounts.recipient.to_account_info(),
    ctx.accounts.mint_authority.to_account_info(),
    signer_seeds,
    voting_event.initial_mint)?;
    //@audit shuuld have been +=
voting_event.total_minted = voting_event.initial_mint;
msg!("Admin minted {} BTR tokens.", voting_event.initial_mint);
```

This bug can cause several issues:

1. If tokens were previously minted to the voting event (e.g., through `approve_mint_event`), this assignment would reset the counter back to only the `initial_mint` amount, effectively "forgetting" about previously minted tokens.
2. Additionally, there's a similar issue a few lines later, where the code checks `voting_event.initial_mint == voting_event.max_btr_mintable` instead of using `total_minted` to determine if the status should be set to `AllMinted`.

Recommendations

Update the code to properly track the total minted amount:

```
mint_tokens(
    ctx.accounts.token_program.to_account_info(),
    ctx.accounts.btr_token.to_account_info(),
    ctx.accounts.recipient.to_account_info(),
    ctx.accounts.mint_authority.to_account_info(),
    signer_seeds,
    voting_event.initial_mint)?;

// Add to total_minted instead of overwriting it
voting_event.total_minted = voting_event.total_minted
    .checked_add(voting_event.initial_mint)
    .ok_or(CustomError::Overflow)?;

msg!("Admin minted {} BTR tokens.", voting_event.initial_mint);
// Mark the voting event as completed
voting_event.status = VotingStatus::Completed;

// Check total_minted against max_btr_mintable
if voting_event.total_minted >= voting_event.max_btr_mintable {
    voting_event.status = VotingStatus::AllMinted;
}
```

These changes ensure that the total minted amount is tracked correctly and the voting event status is updated properly based on the actual total amount of tokens minted.

[L-04] Vault creation blocked if one signer is compromised

Let's take a look at the following code:

```
  if !signers.contains(&ctx.accounts.signer.key()) {
        return Err(ErrorCode::UnauthorizedSigner.into());
    }
```

In order to cancel a request, there is only one key needed to perform such an action. However, when approving the request (for any purpose), all the signers have to sign a tx. This introduces a single point of failure risk where the only compromised signer will be able to cancel all the vault creation requests.

This can also be an issue when the number of signers is equal to the minimum number of signers creating a situation where any new request can't be approved (for example, for updating the signers). To mitigate that, it's also needed to verify that the number of minimum signers needed is less than the total number of signers.

[L-05] Variable naming does not reflect its purpose

Currently, the `voting_event.current_participants` is increased by the number of GOVBTR tokens to burn:

```
voting_event.current_participants += amount;
```

The problem is that it was implemented as a mitigation to one of the findings in the previous audit and here it means "the current number of total votes in GOVBTR tokens" not the current number of participants.

[L-06] Staking duration can be invalid due to prematurely set `staking_end_time`

The `staking_end_time` is set **during request creation**, before the vault event is approved. As a result, the **duration between approval and** `staking_end_time` **is unpredictable**, and in some cases, it may already have **passed** by the time the event is approved. This can lead to critical issues in staking behavior and vault state management.

```
vault_pool.lock_duration = lock_duration;
vault_pool.lock_end_time = 0;
vault_pool.staking_end_time = staking_end_time;
```

Because `staking_end_time` is determined **before** the approval, it may result in:

- **Unexpected or near-zero staking durations**: If approval happens much later than request creation, users may not have sufficient time to stake.
- **Invalid staking period**: If approval happens after `staking_end_time`, the staking window has already expired, rendering staking impossible.
- **Vault freeze**: Since staking can't happen, the vault may remain stuck in the approved state without progressing to further states like locking or distribution.

For example:

- Vault creation: timestamp = 100.
- `staking_end_time` = 500.
- Approval: timestamp = 500 (or later).

In this case, there is **no remaining time** for staking after approval, which defeats the purpose of the staking process.

Recommendations

Defer setting the `staking_end_time` until the vault event is approved. Calculate it dynamically based on the approval timestamp:

```
vault_pool.staking_end_time = Clock::get()?.unix_timestamp + expected_staking_duration;
```

This ensures a consistent and valid staking window relative to the actual approval time, avoiding vault freezes and unexpected behavior.

[L-07] Improper signer validation leads to permanent denial of service

Both programs rely on an authorized list of signers for critical operations, such as creating voting or minting events. However, the mechanism for setting and validating this signer list does not handle certain edge cases, including duplicate entries and insufficient signer uniqueness.

There are two main issues:

1. **Insufficient Unique Signers:** If the number of *unique* signers in the list is less than `min_num_signers`, the protocol may enter a **permanent denial of service** state. Since the required quorum cannot be met, critical operations such as creating Minting , Voting events will remain blocked indefinitely.
2. **Edge Case with Single Signer:** If `min_num_signers == 1`, the following logic leads to an unresolvable DoS:
3. `num_approved_signers` is initialized to 1.
4. It is incremented before checking the approval threshold.
5. Then, the code checks if `min_num_signers` has been met. This flow causes the approval check to behave incorrectly, preventing progress and effectively locking the system.

Here is the related initialization function:

```
pub fn initialize(ctx: Context<Initialize>, signers: [Pubkey; 5], min_num_signers: u8) ->
Result<()> {
    let global_data = &mut ctx.accounts.global_data;
    global_data.admin = ctx.accounts.admin.key();
    global_data.signers = signers;
    global_data.min_num_signers = min_num_signers;
    global_data.total_vaults = 0;
    global_data.usdc_token = ctx.accounts.usdc.key();
    global_data.btr_token = ctx.accounts.btr.key();
    global_data.treasury_account = ctx.accounts.treasury_account.key();
    Ok(())
}
```

Recommendations

- Ensure that the `signers` list contains **no duplicate entries**.
- Assert that the number of **unique signers** is greater than or equal to `min_num_signers`.
- Prevent the edge case where `min_num_signers == 1` and `signers.len() == 1` without proper handling.

Suggested check:

```
let mut unique_signers = Vec::new();
for signer in authorized_signers.iter() {
    if !unique_signers.contains(signer) {
```

```
        unique_signers.push(*signer);
    }
}

require!(
    unique_signers.len() >= min_num_signers as usize,
    CustomError::InvalidSignerConfiguration
);
```

Also, maybe consider adding a minimum threshold to `min_num_signers` to prevent misconfiguration, such as requiring at least 2 signers.

[L-08] Inconsistent TWAP staleness check allows use of stale price data

The codebase defines a constant for TWAP staleness threshold:

```
pub const STALENESS_THRESHOLD: u64 = 60; // staleness threshold in seconds
```

However, in the `normalize_twap` function, the staleness check is hardcoded to 3600 seconds (1 hour):

```
pub fn normalize_twap(&self, decimals_a: u8, decimals_b: u8) -> Result<f64> {
    let raw_twap = self.get_twap_price(3600)?; // @audit 1hr max staleness
    let scale_factor = 10u64.pow((18 + decimals_b as u32).saturating_sub(decimals_a as u32));
    Ok(raw_twap as f64 / scale_factor as f64)
}
```

This is inconsistent with the intended staleness threshold of 60 seconds. As a result, the function may accept and use price data that is up to 1 hour old, even though the rest of the code expects much fresher data.

Recommendations

- Always use the defined `STALENESS_THRESHOLD` constant for all staleness checks, including in `normalize_twap` and any related functions.

[L-09] `VaultPool` and escrow token rent lamports not refunded on cancellation

When a new vault is requested, both the `VaultPool` and its associated escrow `TokenAccount` are created with the `signer` as the payer:

```
#[account(
    init,
    payer = signer, // the payer is mutable
    space = VAULT_POOL_SIZE,
    seeds = [
        VAULT_POOL_PREFIX.as_bytes(),
        &global_data.total_vaults.to_le_bytes()
    ],
```

```
    bump
)]
pub vault_pool: Box<Account<'info, VaultPool>>,
#[account(
    init,
    payer = signer, // the payer is mutable
    token::mint = mint,
    token::authority = vault_pool,
    seeds = [
        TOKEN_ESCROW_PREFIX.as_bytes(),
        &mint.key().to_bytes(),
        &vault_pool.key().to_bytes()
    ],
    bump
)]
pub escrow_token: Box<Account<'info, TokenAccount>>,
```

However, when a vault request is canceled (e.g., `vault.status = VaultStatus::Cancelled;`), these accounts are not closed, and the rent-exempt lamports remain locked in the accounts. There is no logic to close these accounts and refund the rent to the original payer (creator) or anyone else.

But this refund introduces a potential dos attack for cancellation of any request, If any tokens are transferred into the escrow token account (even 1 token), the account cannot be closed until its balance is zero. This allows anyone to grief the system by sending tokens to the escrow, causing a denial of service (DoS) and preventing rent recovery.

Recommendations

- Add a `creator: Pubkey` field to the `VaultPool` struct to record the original payer.
- When canceling a vault request, before closing the escrow token account, sweep any remaining tokens to a designated governance address (e.g., a treasury or admin-controlled account).
- Only attempt to close the escrow account after confirming its token balance is zero. Then close the escrow account, sending rent to the creator.
- Finally, close the `VaultPool` account, sending rent to the creator.

Example fix: ```rust let escrow_balance = escrow_token.amount; if escrow_balance > 0 { // transfer tokens to governance controlled address, make sure escrow_token is empty }

[L-10] `UpdateSignersEvent` rent refund goes to approver not creator

When an `UpdateSignersEvent` account is created, the admin (creator) pays the rent for the account. However, upon approval and closure of the `UpdateSignersEvent`, the rent-exempt lamports are refunded to the last signer who approves the event, not the original creator.

rent goes to approver(not original creator)

```
    let update_event_account_info = update_signer_event.to_account_info();
    let recipient_account_info = ctx.accounts.admin.to_account_info();
    **recipient_account_info.lamports.borrow_mut() += update_event_account_info.lamports();
    **update_event_account_info.lamports.borrow_mut() = 0;
    update_event_account_info.try_borrow_mut_data()?.fill(0);
```

This issue is also present in the vault program, specifically in `vault/programs/btr-vault/src/instructions/approve_update_signer.rs`, where the rent refund is sent to the last approver (signer) instead of the creator. The same remediation applies there too.

Recommendations

- Add a `creator: Pubkey` field to the `UpdateSignersEvent` struct to record the original payer.
- When closing the `UpdateSignersEvent` account after approval, ensure the rent refund is sent to the `creator` account stored in the `UpdateSignersEvent`, not the last approver.

[L-11] `MintEvent` rent refund goes to approver not creator

When a `MintEvent` account is created, the `owner` (creator) pays the rent for the account. However, upon approval and closure of the `MintEvent`, the rent-exempt lamports are refunded to the last signer who approves the event, not the original creator. This is because the code explicitly transfers the lamports to the signer (approver) instead of the creator who funded the account creation.

The issue is visible in the `MintEvent` struct and the account initialization in `admin_functions.rs`:

```
#[account(
    init,
    payer = owner,
    space = MINT_EVENT_SIZE,
    seeds = [b"create_mint_request", &global_data.request_id.to_le_bytes()[..]],
    bump
)]
pub mint_event: Account<'info, MintEvent>,
```

But there is no `creator` field in `MintEvent`, and the rent refund is sent to the last approver, not the creator. Thus, the rent refund does not go to the correct party.

Recommendations

- Add a `creator: Pubkey` field to the `MintEvent` struct to record the original payer.
- When closing the `MintEvent` account after approval, ensure the rent refund is sent to the `creator` account stored in the `MintEvent`, not the last approver.

[L-12] Missing recipient validation for voting event and associated Mint event

When minting tokens associated with a voting event (indicated by `is_voting_from != 0`), there's no validation that the recipient address matches the intended recipient from the voting event. This allows an authorized signer to create mint events that direct tokens intended for a voting event participant to any arbitrary address.

The vulnerability exists in both the `create_mint_event` and `approve_mint_event` functions, where tokens can be minted to an arbitrary recipient while still counting toward the voting event's total allocation (`total_minted`).

This creates a critical issue because:

1. The `total_minted` counter for the voting event is still incremented.
2. When `total_minted == max_btr_mintable`, the voting event status is changed to `VotingStatus::AllMinted`.
3. Once in this state, no further tokens can be minted for the legitimate recipient of the voting event.

Here's the problematic code from `approve_mint_event`:

```
if mint_event.is_voting_from != 0 {
    voting_event.total_minted = voting_event
        .total_minted
        .checked_add(mint_event.amount)
        .ok_or(CustomError::Overflow)?;

    if voting_event.total_minted == voting_event.max_btr_mintable {
        voting_event.status = VotingStatus::AllMinted;
    }
}
```

The function updates the voting event state without verifying that the recipient (`mint_event.receiver_address`) is the intended recipient for that voting event.

Recommendations

Add validation checks in both `create_mint_event` and `approve_mint_event` to ensure that when a mint event is associated with a voting event, the recipient address matches the voting event's intended recipient:

1. In `create_mint_event`:

```
if is_voting_from != 0 {
    let voting_event = &mut ctx.accounts.voting_event;
    // Ensure the voting event is completed
    if voting_event.status != VotingStatus::Completed {
        return Err(CustomError::InvalidVotingStatus.into());
    }
```

```
    if voting_event.total_minted + amount > voting_event.max_btr_mintable {
        return Err(CustomError::ExceedsMintLimit.into());
    }

    // Add this check to validate recipient matches voting event recipient
    if receiver_address != voting_event.recipient {
        return Err(CustomError::InvalidRecipient.into());
    }
}
```

1. Add the new error type to the `CustomError` enum:

```
#[error("Invalid recipient for voting event")]
InvalidRecipient,
```

[L-13] Missing seed validation allows unauthorized token approval

In `admin_functions.rs` the `approve_mint_event` function fails to properly validate that the provided `global_data` account is the legitimate PDA derived from legitimate seeds. Instead, it only enforces the account type without verifying it is derived using proper seeds and canonical bumps. Anyone can craft a fake `GlobalData` pda set its owner to our program and pass it here, only validation that's happening here is that account is owned by our program and has `GlobalData` structure/discriminator. They are not checking for seeds at all, unlike the other places in code where they are correctly making sure that this account is a legitimate pda derived from proper seeds.

```
pub struct ApproveMintEvent<'info> {
    // ...other accounts...
    //@audit seeds not validated,
    //  1. can dos a voting event
    //  2. can approve all mint events(evenn the ones that got unapproved in past, can pass
those accounts and approve them)
    pub global_data: Account<'info, GlobalData>,
    // ...other accounts...
}
```

Unlike other PDAs in this struct that use the `seeds` and `bump` constraints, the `global_data` account lacks these validations.

An attacker could: 1. Create a malicious program that initializes a `GlobalData` account with the same structure and sets our program as program owner. 2. Set themselves as the only authorized signer in the `authorized_signers` array. 3. Set `min_num_signers` to 1. 4. Pass this spoofed account as the `global_data` parameter when calling `approve_mint_event`. 5. Approve mint events single-handedly, even past events that were not previously approved. 6. He can calculatingly mint the exact amount of tokens to satisfy the below condition, which will leave a voting event in `VotingStatus::AllMinted` status,

```
if voting_event.total_minted == voting_event.max_btr_mintable {
    voting_event.status = VotingStatus::AllMinted;
}
```

1. thus this voting event is useless, users can't vote here as to vote, it requires the voting event to be live, so this way it corrupts genuine voting events as well.

```
    require!(voting_event.status == VotingStatus::Live, CustomError::VotingEventNotLive);
```

This would effectively allow unauthorized token minting and denial-of-service attacks against legitimate voting events by prematurely setting their status to `AllMinted`.

Recommendations

Add proper seed validation for the `global_data` account in the `ApproveMintEvent` struct:

```
#[derive(Accounts)]
pub struct ApproveMintEvent<'info> {
    // ...other accounts...
    #[account(
        seeds = [GLOBAL_DATA_PREFIX.as_bytes()],
        bump
    )]
    pub global_data: Account<'info, GlobalData>,
    // ...other accounts...
}
```

This ensures that only the legitimate global data PDA can be used in this context, preventing attackers from bypassing authorization checks with spoofed accounts.