#HashEx
BLOCKCHAIN SECURITY

Fortuva

smart contracts final audit report

July 2025

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the **Fortuva** team to perform an audit of their smart contract. The audit was conducted between **2025-07-08** and **2025-07-30**.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code was provided in a zip file.

SHA256 hashes of audited files:

`d73910ac13e133c1c940b6858113b85880e694aa3851057d652f39f105bbacb0` lib.rs,

`46cfc1d48074a670014b6b4adedcecad96d8778a259141f90fb711d7680971df` instructions/cancel_bet.rs,

`b0976a5bc05e9eabcf31ca6ca61ba771c1144cebd10e32e5e60a1120475aece1` instructions/claim_payout.rs,

`e9c30e1e7f6361b59a4b3456e21f666daa03793a40ddfb2cfc4c48c931ec4ee5` instructions/close_round.rs,

`79916e36bfffe2c947df5d9a47269a57a88b57a2916e9fbd68fb1de105deb247` instructions/end_start_round.rs,

`2444f3c290fd22aa7bb290dc46a4b6d8f0e09c5410b3e923a58de6065156915a` instructions/init.rs,

`ff81f735d3860f7334b9591412d10a17d5e24dee49496be1fd8cb1512c97641f` instructions/

lock_round.rs,

`f3aa60a48011017e0538f603b5ac53f2848ea62619823cc4c59f7a287858573e` instructions/place_bet.rs,

`e002bdc86babd81231791d177ea0cc9c6f631a032d85cf9874bc62cc78718938` instructions/start_genesis.rs,

`f3028e24f6764d1e21b066a248cd698f909564b428ac0f50b5a42c93af8a539e` instructions/update_config.rs,

`9c73726f9c2d5501ae980cc2b0569c04e6aae91a7a30cd952cbb7b154b6f86bb` instructions/update_multisig.rs,

`e81c6f28b23d060af25fd9e7694169acd9caf2308d6a524013aa8972610fcbc1` instructions/withdraw_treasury.rs,

`a1acd14052dfec69e55f13d3cefabb820ba25b275497a37d64928f842aaf26bc` instructions/mod.rs,

`e364c511fd0a2b31c4ddadf9f8a960737aaaba6a8bf9a3cb6a98f32c45175ebb` events/mod.rs,

`6734bed38b7ca79eb2b87ba9532ea5e6e6b1148c8c9b3adca58060caa2fe52f9` errors/mod.rs,

`7882e6493522e79b893b9360e772587dfd5bf596a133b075f282cfd6060c8142` state/mod.rs,

`d6e4743248675b1ba895423739798ac0728f253021306f1a519aaa7184ef909b` state/config.rs,

`2fbd6a580f063d1f6335deccbdb43713d597319a98bfc6fefe5280bcc29251f7` state/round.rs,

`1d5db40bf69b78514052ab18836aa306ac31b88cf7e0e59142d30236558e29bb` state/user_bet.rs.

SHA256 hashes of updated files:

`6f6b4b2e0e0c135ad64d39230bceeb0d3f34b68e4ab0e7a793290a7999119feb` lib.rs,

`405297b9affe39a9144df1466d8c1334360d278aa914916afa7ec653d8f78695` instructions/

cancel_bet.rs,

ca61706a583b658d62de05ac96f9a251e6b3c9791553c62e37a0bcfa32ee9f9f instructions/claim_payout.rs,

c5faf198cdd2928bc813400e1fbbd935659cfb486065b60cce9c5d6da679a049 instructions/close_round.rs,

ef4148e42ef72a68370afcfe37c34489412b8eedbbb24c63d03fd8ebb3d02f80 instructions/end_start_round.rs,

9c76aeb9f981f157f5eda386e5888c1a7aa3a527b32863af9f7402d54e85cac4 instructions/init.rs,

13badfb5bc33d0ac015680ec7b17578d87c4ef398d36da4ebe7be0e472c5959f instructions/lock_round.rs,

dbbb5d6e66ae08e7aa35013f8843bb0228a7aa0e54b1e22b3ad5cd42d3e97f9b instructions/place_bet.rs,

673c37294886fdd24f83d70fbf3db458b1b1e4f6ab466f047c43a20ea31ddf2a instructions/start_genesis.rs,

81a295c13759437b49df31df8f4eea3e5ad6e239365870091dd7b22e92c24156 instructions/update_config.rs,

5617d7b6457079f4f8c493e0ee9be332697918c4a11d5d649d882671472305b6 instructions/update_multisig.rs,

1f006b1f7c98d269803c3ab5b2e3a24933ca4f61ee50c4efbac3ee812d1c9982 instructions/withdraw_treasury.rs,

a1acd14052dfec69e55f13d3cefabb820ba25b275497a37d64928f842aaf26bc instructions/mod.rs,

048cb3f72d424884aa94ccea524b83d17ea33bfbc7a2648df774783a5255fb0d events/mod.rs,

a5aab80be6cff2519337a24d2e988bb672351a0f51aa85b1e468fa745e356867 errors/mod.rs,

7882e6493522e79b893b9360e772587dfd5bf596a133b075f282cfd6060c8142 state/mod.rs,

77cdfc607453942fc7d8e8c7ffe5662bfbf11b679e4b52fee376658b25e5d71f state/config.rs,

46e4aeb8822d15829b5f2c2b8b2533bc77f067c7480a833460bad4febfa66a8c state/round.rs,

1d5db40bf69b78514052ab18836aa306ac31b88cf7e0e59142d30236558e29bb state/user_bet.rs.

2.1 Summary

Project name	Fortuva
URL	https://app.fortuva.xyz
Platform	Solana
Language	Rust
Centralization level	High
Centralization risk	Medium

2.2 Contracts

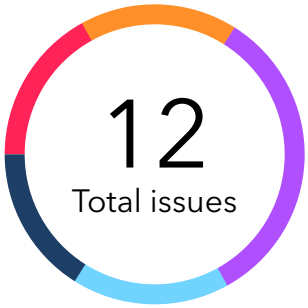
Name	Address
lib.rs	

3. Project centralization risks

The owner can update configuration to set up executor account, fee percent, pause state, round parameters. The time-related parameters may be misconfigured to prevent any round from successful finalization, meaning that all rounds may ended cancelled until the owner fixes the config parameters `round_duration`, `lock_duration`, `buffer_seconds`.

The executor can abstain from finalizing prediction rounds. The executor can manipulate prediction result within the buffer period from configuration.

4. Found issues



Critical	2 (17%)
High	2 (17%)
Medium	4 (33%)
Low	2 (17%)
Info	2 (16%)

Cc3. lib.rs

ID	Severity	Title	Status
Cc3I7c	Critical	Single signer can bypass multisig threshold	Resolved
Cc3I7d	Critical	Lack of address checks in update multisig instruction	Resolved
Cc3I78	High	Mutisig account resizing	Resolved
Cc3I7b	High	Incorrect oracle price calculation	Resolved
Cc3I82	Medium	Executor can influence results	Partially fixed
Cc3I7a	Medium	Logic error may prevent round closure	Resolved
Cc3I81	Medium	Allowed bets for future round	Resolved
Cc3I79	Medium	cancel_bet instruction does not update treasury.amount	Resolved
Cc3I7e	Low	Unused config account in claim payout instruction	Resolved

Cc3I83	Low	Duplicated code	Resolved
Cc3I7f	Info	Naming inconsistency	Resolved
Cc3I80	Info	Pause state is ignored for claiming	Acknowledged

5. Contracts

Cc3. lib.rs

Overview

The program implements a binary options prediction market with the following key actors and lifecycle:

- Admin: A multisig-controlled authority responsible for initializing the contract, updating critical operator configurations, and withdrawing treasury profits.
- Operator: A multisig-controlled authority for configuration changes (e.g., fees, durations).
- Executor: A designated hot-wallet/bot responsible for automating round transitions. It locks rounds, ends previous rounds, and starts new ones based on time and Pyth oracle data.
- User: An end-user who can place bets on "Bull" (price up) or "Bear" (price down) outcomes and claim rewards if they win.

Round Lifecycle:

1. initialize: The contract is set up by the admin.
2. start_genesis_round: The executor starts the first round.
3. place_bet: Users bet on the currently active round before its lock_time.
4. lock_round / end_and_start_round: At the scheduled time, the executor fetches a price from Pyth to:Lock the current round's price.Settle the previous round (determine winner, calculate rewards).Start the next round.
5. claim_payout: Winning users claim their share of the prize pool from the settled round.
6. close_round: Once all winners from a round have claimed, the executor can close the round's account to reclaim its rent.

Issues

Cc3I7c Single signer can bypass multisig threshold

Critical Resolved

The `verify_multisig()` helper instruction, used by all administrative instructions (`update_config`, `withdraw_treasury`, `update_multisig`), contains a critical flaw in its signature verification logic. It iterates through the `remaining_accounts` provided in a transaction and increments a counter for each valid signature. However, it fails to check for duplicate signers.

This allows a single malicious signer, who is part of the multisig set, to bypass the required threshold by including their own public key multiple times in the transaction's `remaining_accounts`. This completely undermines the security of the multisig, allowing one malicious actor to unilaterally execute protected administrative functions.

```
fn verify_multisig<'info>(multisig: &Multisig, signers: &[AccountInfo<'info>]) ->
Result<()> {
    let mut valid_signatures = 0;

    for signer in signers.iter() {
        if signer.is_signer && multisig.signers.contains(signer.key) {
            valid_signatures += 1;
        }
    }

    require!(
        valid_signatures >= multisig.threshold,
        PoolErrorCode::NotEnoughSignatures
    );

    Ok(())
}
```

Recommendation

The `verify_multisig()` instruction must be modified to ensure that each unique signer is only counted once.

Cc3I7d Lack of address checks in update multisig instruction

Critical Resolved

The `update_multisig()` instruction allows the `admin_multisig` to change the signers and threshold of the `operator_multisig`. However, the instruction's context `UpdateMultisig` does not enforce any relationship between the provided `config`, `operator_multisig`, and `admin_multisig` accounts.

```
#[derive(Accounts)]
pub struct UpdateMultisig<'info> {
    #[account()]
    pub config: Account<'info, Config>,
    #[account(mut)]
    pub operator_multisig: Account<'info, Multisig>,
    #[account(mut)]
    pub admin_multisig: Account<'info, Multisig>,
}
```

An attacker can create a transaction that passes a legitimate `operator_multisig` account but provides fake, attacker-controlled `config` and `admin_multisig` accounts. Because the function only checks that the public key of the fake `admin_multisig` matches a field in the fake config, and that the `operator_multisig` key matches another field, all checks pass. The attacker can then use their own fake `admin_multisig` (for which they have all the "required" signatures) to take control of the real `operator_multisig` account.

```
pub fn update_opmultisig(
    ctx: Context<UpdateMultisig>,
    new_signers: Vec<Pubkey>,
    new_threshold: u8,
) -> Result<()> {
```

```
    let config = &mut ctx.accounts.config; //@note Do I need config here? Is mutablity
required?
    let op_multisig = &mut ctx.accounts.operator_multisig;
    let ad_multisig = &mut ctx.accounts.admin_multisig;

    require!(config.operator_multisig == op_multisig .key(),
PoolErrorCode::InvalidMultisig);
    require!(config.admin_multisig == ad_multisig.key(), PoolErrorCode::InvalidMultisig);
    ...
}
```

Recommendation

Ensure that the address of the config PDA passed to the instruction is exactly the same as was initialized in the `initialize()` instruction.

Cc3I78 Mutisig account resizing

High Resolved

The Initialize instruction allocates space for the `admin_multisig` and `operator_multisig` accounts based on the length of the signers list passed in the instruction arguments.

```
#[derive(Accounts)]
#[instruction(round_duration: u64, min_bet_amount: u64, treasury_fee: u64, lock_duration:
u64, buffer_seconds: u64, executor: Pubkey, signers: Vec<Pubkey>, threshold: u8)]
pub struct Initialize<'info> {
    #[account(
        init,
        payer = admin,
        space = 8 + size_of::<Config>(),
        seeds = [b"config"],
        bump
    )]
    pub config: Account<'info, Config>,
    #[account(
        init,
        payer = admin,
        space = 8 + size_of::<Treasury>(),
        seeds = [b"treasury"],
        bump
```

```
    )]
    pub treasury: Account<'info, Treasury>,
    #[account(
        init,
        payer = admin,
        space = 8 + 4 + signers.len() * 32 + 1 + 1,
        seeds = [b"adminmultisig"],
        bump
    )]
    pub admin_multisig: Account<'info, Multisig>,
    #[account(
        init,
        payer = admin,
        space = 8 + 4 + signers.len() * 32 + 1 + 1,
        seeds = [b"operatormultisig"],
        bump
    )]
    pub operator_multisig: Account<'info, Multisig>,
    #[account(mut)]
    pub admin: Signer<'info>,
    pub system_program: Program<'info, System>,
}
```

However, the `update_multisig()` function allows the admin to change signers of the operator multisig. If the size of the signers list is bigger than the initial signers number, this will lead to data corruption.

Recommendation

Allocate space for `MAX_SIGNERS` and check that the new list if signers is always less than the `MAX_SIGNERS`.

Cc3I7b Incorrect oracle price calculation

High Resolved

Prediction results are inquired from the Pyth price feed and stored in the program data by calling either the `lock_round()` or `end_and_start_round()`, both available only for executor role. The Pyth price feed returns price in the exp notation.

The `lock_round()` instruction converts that float price to u64 number, potentially losing the

precision. In worst case the resulting price can be calculated as zero, breaking further safety checks.

The `end_and_start_round()` instruction includes fixed multiplier 1e8 without checking if the price exp is lower than 8.

Moreover, if the round is locked via the `lock_round()` instruction and then the `end_and_start_round()` is called, the stored price without multiplier is compared with the multiplied one, resulting in completely false outcome.

```
    let price_update = &ctx.accounts.pyth_price;
    let maximum_age: u64 = 30;
    let feed_id: [u8; 32] = get_feed_id_from_hex(
        "0xef0d8b6fda2ceba41da15d4095d1da392a0d2f8ed0c6c7bc0f4cfac8c280b56d"
    )?;
    let price = price_update.get_price_no_older_than(&Clock::get()?, maximum_age,
&feed_id)?;

    let current_price = if price.exponent >= 0 {
        (price.price as u64)
            .checked_mul((10u64).pow(price.exponent as u32))
            .ok_or(PoolErrorCode::MathOverflow)?
    } else {
        (price.price as u64)
            .checked_div((10u64).pow(price.exponent.abs() as u32))
            .ok_or(PoolErrorCode::MathOverflow)?
    };

    round.lock_price = current_price;
    emit!(RoundLockedEvent {
        round_number: round.number,
        price: current_price,
        timestamp: now,
    });
```

Recommendation

Use the same multiplier for both instructions. Include safety checks comparing multiplier with price exp.

Cc3I82 Executor can influence results

Medium Partially fixed

The executor account is responsible for locking the round price with `lock_round()` and for advancing the rounds with `end_and_start_round()`. The problem is although the round's lock time and close time are determined, the executor can call the program at any moment after that stored round deadlines. The prices are locked not for the round deadline timestamps but for the transaction timestamp, so the executor can pospone his calls to increase the chance of desired result.

```
pub fn end_and_start_round(ctx: Context<EndAndStartRound>) -> Result<()> {
    let config = &mut ctx.accounts.config;
    let previous_round = &mut ctx.accounts.previous_round;
    let current_round = &mut ctx.accounts.current_round;
    let new_round = &mut ctx.accounts.new_round;
    let treasury = &mut ctx.accounts.treasury;

    require!(ctx.accounts.executor.key() == config.executor,
PoolErrorCode::UnauthorizedExecutor);
    require!(!config.is_paused, PoolErrorCode::ContractPaused);
    require!(config.current_round > 0, PoolErrorCode::GenesisRoundNotStarted);
    require!(config.genesis_locked, PoolErrorCode::GenesisRoundNotLocked);
    require!(current_round.is_active, PoolErrorCode::RoundNotActive);
    let now = Clock::get()?.unix_timestamp;
    require!(now >= previous_round.close_time, PoolErrorCode::RoundNotClosed);
    require!(previous_round.lock_price != 0, PoolErrorCode::RoundNotLocked);

    let price_update = &ctx.accounts.pyth_price;
    let maximum_age: u64 = 30;
    let feed_id: [u8; 32] = get_feed_id_from_hex(
        "0xef0d8b6fda2ceba41da15d4095d1da392a0d2f8ed0c6c7bc0f4cfac8c280b56d"
    )?;
    let price = price_update.get_price_no_older_than(&Clock::get()?, maximum_age,
&feed_id)?;
```

```
    ...
}
```

Recommendation

Consider getting the price for the stored round deadline instead of transaction timestamp.

Update

A buffer has been implemented, during which the executor must set the `lock_price` or `close_price` of the round. Consequently, the executor can now only wait for a desired price within this buffer period. In a configuration where the buffer time is significantly shorter than the round's duration, this substantially limits the executor's ability to manipulate the outcome of the round.

Severity of the issue was reduced from high to medium one.

Cc3I7a Logic error may prevent round closure

Medium Resolved

The `close_round()` instruction is used to close round's account after all winners have claimed their rewards. It checks if number of claimed users is equal to the number of winners. However, there is copy-paste error in the logic: when end price is less than the lock price, it incorrectly checks `round.total_bull_count` instead of `round.total_bear_count`.

```
pub fn close_round(ctx: Context<CloseRound>) -> Result<()> {
  ...

  if round.end_price > round.lock_price {
    require!(round.total_bull_count == round.claimed_count,
PoolErrorCode::RoundNotClosed);
  }
  else if round.end_price < round.lock_price {
    require!(round.total_bull_count == round.claimed_count,
PoolErrorCode::RoundNotClosed);
  }

  ...
```

```
}
```

Cc3I81 Allowed bets for future round

Medium Resolved

Users are allowed to call the `place_bet()` function with the round number equal to the current round store in the config or the `current+1`. The problem is the start of the next round is not predefined at the moment of placing such bet. Round switching is controlled by the executor role and the only constraint is that the next round can't precede the `close_time` of the previos one.

```
pub fn place_bet(
    ctx: Context<PlaceBet>,
    amount: u64,
    predict_bull: bool,
    round_number: u64,
) -> Result<()> {
    let config = &ctx.accounts.config;
    let round = &mut ctx.accounts.round;
    let user_bet = &mut ctx.accounts.user_bet;
    let treasury = &mut ctx.accounts.treasury;

    require!(!config.is_paused, PoolErrorCode::ContractPaused);
    require!(round.is_active, PoolErrorCode::RoundNotActive);
    let now = Clock::get()?.unix_timestamp;

    require!(
        round.number == config.current_round || round.number == config.current_round + 1,
        PoolErrorCode::InvalidRound
    );
    require!(round.number == round_number, PoolErrorCode::InvalidRound);
    require!(round.lock_price == 0, PoolErrorCode::RoundLocked);
    require!(now < round.lock_time, PoolErrorCode::RoundLocked);
    require!(amount >= config.min_bet_amount, PoolErrorCode::BetBelowMinimum);
    require!(user_bet.amount == 0, PoolErrorCode::AlreadyBetInRound);
    ...
}
```

Recommendation

Provide sufficient information to users about possibility of freezing bets or remove bets for the next round.

Cc3I79 cancel_bet instruction does not update treasury.amount

Medium Resolved

The program maintains the treasury balance in two places:

1. The actual lamports held by the treasury PDA.
2. An internal counter treasury.amount within the treasury account's data.

Most functions like `place_bet()` and `withdraw_treasury()` correctly update both. However, the `cancel_bet()` function fails to do so. It transfers lamports from the treasury PDA back to the user but does not decrement the `treasury.amount` field.

```
pub fn cancel_bet(ctx: Context<CancelBet>, round_number: u64) -> Result<()> {
    let config = &ctx.accounts.config;
    let round = &mut ctx.accounts.round;
    let user_bet = &ctx.accounts.user_bet;
    let treasury = &mut ctx.accounts.treasury;

    require!(config.is_paused, PoolErrorCode::ContractNotPaused);
    require!(round.number == round_number, PoolErrorCode::InvalidRound);
    require!(
        user_bet.round_number == round_number,
        PoolErrorCode::InvalidRound
    );
    require!(!user_bet.claimed, PoolErrorCode::AlreadyClaimed);

    require!(treasury.amount >= user_bet.amount,
PoolErrorCode::InsufficientTreasuryFunds);

    let amount = user_bet.amount;
    **treasury.to_account_info().try_borrow_mut_lamports()? -= amount;
    **ctx
        .accounts
```

```
        .user
        .to_account_info()
        .try_borrow_mut_lamports()? += amount;

    if user_bet.predict_bull {
        round.total_bull_amount = round
            .total_bull_amount
            .checked_sub(amount)
            .ok_or(PoolErrorCode::MathOverflow)?;
        round.total_bull_count = round
            .total_bull_count
            .checked_sub(1)
            .ok_or(PoolErrorCode::MathOverflow)?;
    } else {
        round.total_bear_amount = round
            .total_bear_amount
            .checked_sub(amount)
            .ok_or(PoolErrorCode::MathOverflow)?;
        round.total_bear_count = round
            .total_bear_count
            .checked_sub(1)
            .ok_or(PoolErrorCode::MathOverflow)?;
    }

    ctx.accounts.user_bet.close(ctx.accounts.user.to_account_info())?;

    emit!(BetCanceled {
        user: ctx.accounts.user.key(),
        round_number,
        amount,
        timestamp: Clock::get()?.unix_timestamp,
    });
    Ok(())
}
```

Recommendation

Update treasury amount in the `cancel_bet()` instruction or remove tracking treasury amount at all.

Cc3I7e Unused config account in claim payout instruction

Low Resolved

The config account is passed to the `claim_payout()` instruction, but is never read or used. We recommend removing the unused account.

Cc3I83 Duplicated code

Low Resolved

The `end_and_start_round()` function contains second safety check for round's state.

```
require!(current_round.is_active, PoolErrorCode::RoundNotActive);
```

Cc3I7f Naming inconsistency

Info Resolved

The block in lib.rs exposes a function named `update_mulitisig()`, but the implementation in the update_multisig.rs uses `update_opmultisig()`.

Cc3I80 Pause state is ignored for claiming

Info Acknowledged

The program contains pausing option in its config, which is checked for the most of functions. However, the `claim_payout()` function allows claiming even in the paused state.

6. Conclusion

2 critical, 2 high, 4 medium, 2 low severity issues were found during the audit. 2 critical, 2 high, 3 medium, 2 low issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter. This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. Centralization risks classification

Centralization level

High. The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.

Medium. The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.

Low. The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

High. Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.

Medium. Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.

Low. Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY