SHERLOCK

Security Review For matrix.fun



Collaborative Audit Prepared For: **matrix.fun**
Lead Security Expert(s): **ArmedGoose**
JakubHeba
Date Audited: **November 19 - November 23, 2025**

Introduction

Matrix.fun is an onchain engagement engine that allows projects to host no-code participation campaigns using their own native token as the entry and reward mechanism. Users join these campaigns, across AAA games, casual games, trading competitions, and other interactive activities, by transacting directly in the project's token. Each event is deployed as an independent smart contract, ensuring transparent entries, rewards, and verifiable onchain actions. This structure significantly reduces bot participation and provides ecosystems with scalable, high-signal user engagement.

Scope

Repository: matrix-fun-ag/smart-contracts

Audited Commit: 27092fa47dc674fcfe1a00ba1677217c37ee76ed

Final Commit: 308977d5d97126ad943caba972b49d45bf494959

Files:

- sources/config.move
- sources/token_handler.move
- sources/tournament.move

Final Commit Hash

308977d5d97126ad943caba972b49d45bf494959

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
4	5	10

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Public access control allows manipulation of internal balance accounting [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/4

Summary

In `smart-contracts/sources/token_handler.move`, functions `increase_balance` and `decrease_balance` are public and allow to modify `TokenBalance` entries for arbitrary `resource_addr` argument. Any user can cause temporary denial of service by artificially decreasing balance.

Vulnerability Detail

The `token_handler` module tracks internal balances for tokens held by the contract. These accounting functions lack proper access control, enabling any malicious contract to call `decrease_balance` and set token balances to zero, even when the contract holds real assets. This creates a mismatch between actual holdings and internal accounting. While increasing balance has no real impact on funds, this should also be restricted.

Impact

Malicious users may call `decrease_balance` without restrictions, blocking withdrawals that rely on accounted balance.

Code Snippet

In `smart-contracts/sources/token_handler.move`:

```
/// Decrease balance
public fun decrease_balance(
    resource_addr: address,
    token_address: address,
    amount: u64,
) acquires TokenBalance {
    let token_balance = borrow_global_mut<TokenBalance>(resource_addr);

    assert!(table::contains(&token_balance.balances, token_address),
    ↪  E_INSUFFICIENT_BALANCE);
    let current_balance = table::borrow_mut(&mut token_balance.balances,
    ↪  token_address);
    assert!(*current_balance >= amount, E_INSUFFICIENT_BALANCE);
    *current_balance = *current_balance - amount;
}
```

While other administrative functions use `signer` as the address key, for instance:

```
 /// Remove supported token
    public entry fun remove_supported_token(
        resource_account: &signer,
        token_address: address,
    ) acquires SupportedTokens {
        let resource_addr = signer::address_of(resource_account);
[...]
```

Tool Used

Manual Review

Recommendation

Either restrict accessibility to `public(friend)` and set the `token_handler` module as `friend` or use similar `signer` address derivation instead of arbitrary address.

Discussion

magj2006

fix: remove public for both decrease_balance function and increase_balance function, and change public(friend) for remove_supported_token.

Issue H-2: Denial of service in `tournament_finish` due to strict equality requirement [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/10

Summary

The `tournament_finish` function compares the contract's Net prize pool against a Gross expected total.

Vulnerability Detail

The `tournament.prize_pool` variable, calculated in `config::calculate_total_prize_pool` and increased each time new participant joins the pool represents the funds available for participants *after fees have been deducted*.

However, the `tournament_finish` function calculates `expected_total` by adding the fees back onto the distributed prizes. The subsequent assertion `prize_pool == expected_total` effectively checks if `Net_Pool == Net_Pool + Fees`.

Impact

Inability to finish most of the tournaments by design.

Code Snippet

In config::calculate_total_prize_pool:219:

```
let total_prize_pool = initial_prize_contribution + entry_fees_contribution -
↪  host_fees_amount;
```

However in `tournament.move::finish_tournament`:

```
// Calculate total conservation check
let expected_total = total_distributed_prizes + host_remaining_initial_prize +
↪  war_chest_fee_amount + host_fees_amount;
assert!(tournament.prize_pool == expected_total, E_BALANCE_MISMATCH);
```

The same problem may occur, even of those fees are accounted for, but the final amounts will differ by dust amounts due to rounding. The strict equality is difficult to be met due to this. Since that assertion cannot be satisfied, it leads to inability to finish most of the tournaments.

Tool Used

Manual Review

Recommendation

Remove fee amounts from the assertion logic. The assertion should only verify that the distributed prizes do not exceed the available net prize pool.

Discussion

magj2006

Explain: It's correct because prize_pool is sum of initialize_prize_contribution and entry_fee. you can view below picture to know the calculate relation of them.

Issue H-3: Multiple refund possible in `refund_user` function [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/12

Summary

The `refund_user` function allows participants to claim refunds multiple times for the same tournament entry because it lacks any tracking mechanism for completed refunds.

Vulnerability Detail

The `refund_user` function in `tournament.move` verifies that a tournament has `TOURNAMENT_STATUS_CANCELLED` status and that the caller's address exists in the `tournament.participants` vector, then proceeds to transfer the `entry_fee` amount to the user via `token_handler::transfer_fa_from_contract_to_user`.

However, after successfully issuing the refund, the function neither removes the user from the participants vector nor records the refund in any tracking structure. This stands in contrast to the `claim_prize` function which properly uses the `claimed_tournaments` vector in `UserParticipation` to prevent double claims.

Since the participants list remains unchanged and no refund tracking exists, the same user can call `refund_user` repeatedly with valid admin signatures, and each call will pass all validation checks and issue another refund.

Impact

A malicious user can drain all tournament funds by repeatedly calling `refund_user`, receiving the `entry_fee` amount multiple times until the contract's token balance is drained.

Code Snippet

Tool Used

Manual Review

Recommendation

Add a `refunded_tournaments` vector to the `UserParticipation` struct and check against it before processing refunds, then add the `tournament_id` to this vector after successful refund completion.

Discussion

magj2006

Fixed: add refunded_tournaments list to hold refunded user.

Issue H-4: Signature replay attack due to missing message context validation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/14

Summary

The `verify_admin_signature` function validates cryptographic signatures without verifying the semantic content of the signed message, enabling signatures to be reused across different operations, tournaments, and users.

Vulnerability Detail

The `verify_admin_signature` function accepts a signature vector and a message vector, then uses `ed25519::signature_verify_strict` to confirm the signature was created by the delegate's private key for the given message bytes. The function treats the message as an opaque byte array and performs no parsing or validation of its contents.

This means the contract has no way to ensure that a signature authorizing user A to join tournament 1 cannot be reused to authorize user B to join tournament 2, or even to authorize a completely different operation like `leave_tournament` or `claim_prize`.

Furthermore, there is no timestamp or expiration field validated within the message, so signatures remain valid indefinitely. The contract also lacks `nonce` tracking, meaning even a properly-formatted message could be replayed if the same operation needs to be performed again.

Impact

Attackers who obtain any valid admin signature can replay it across arbitrary tournaments, users, and operations, completely bypassing the intended authorization controls and potentially draining funds or manipulating tournament outcomes.

Code Snippet

Tool Used

Manual Review

Recommendation

Implement structured message parsing within `verify_admin_signature` that validates operation type, `tournament_id`, user address, expiration timestamp, and a unique nonce that is tracked on-chain to prevent replay.

Discussion

magj2006

Fixed: Implement structured message parsing within verify_admin_signature that validates operation type, tournament_id, user address, expiration timestamp, and a unique nonce that is tracked on-chain to prevent replay

Issue M-1: Tournament can be stuck in `CREATED` status when full before start time [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/5

Summary

If a tournament reaches `max_players` before `open_time`, the status transition logic in `join_tournament` never executes, leaving the tournament permanently stuck in `CREATED` status.

Vulnerability Detail

The tournament status only transitions from `CREATED` to `ONGOING` in the `join_tournament` function:

```
if ((tournament.status == TOURNAMENT_STATUS_CREATED || tournament.status ==
↪  TOURNAMENT_STATUS_PENDING) && current_time >= tournament.open_time) {
    tournament.status = TOURNAMENT_STATUS_ONGOING;
};
```

If a tournament reaches `max_players` before `open_time`, no more joins can occur, so this transition never executes.

Impact

If that happens, then the only way to receive funds back is to terminate current tournament and refund which causes temporary freeze of funds and may block core functionality of the protocol. Then a new tournament has to be created.

Tool Used

Manual Review

Recommendation

Add a separate routine that allow caller to advance the tournament status if all conditions for `TOURNAMENT_STATUS_ONGOING` are met.

Discussion

magj2006

Fixed: add

```
assert!(current_time >= tournament.open_time, E_INVALID_TIME);
```

```
assert!(
            tournament.status == TournamentStatus::CREATED ||
            tournament.status == TournamentStatus::PENDING ||
            tournament.status == TournamentStatus::ONGOING,
            E_INVALID_STATUS
        );
```

to prevent user join tournament when tournament is not open or on other status.

Issue M-2: Multiple refund in `refund_host` function [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/13

Summary

The `refund_host` function does not reset the `initial_prize_contribution` field to zero after issuing a refund, allowing hosts to potentially claim refunds multiple times.

Vulnerability Detail

When `refund_host` is called, the function reads the `tournament.initial_prize_contribution` value into `refund_amount`, validates that this amount is greater than zero, subtracts it from `tournament.prize_pool`, and transfers the funds to the host address. The flaw is that while `prize_pool` is decremented, the `initial_prize_contribution` field itself is never set to zero.

This means the `refund_amount` variable will contain the same non-zero value on subsequent calls. Although the `prize_pool` subtraction would eventually cause an underflow error when `prize_pool` becomes insufficient, in scenarios where `prize_pool` contains additional funds from entry fees, the host could extract more than their original contribution before the underflow protection triggers.

Impact

A tournament host could extract funds exceeding their initial prize contribution by calling `refund_host` multiple times, effectively stealing entry fees that belong to participants.

Code Snippet

Tool Used

Manual Review

Recommendation

Add the statement `tournament.initial_prize_contribution = 0` immediately after reading the `refund_amount` to ensure subsequent calls will fail the greater-than-zero assertion.

Discussion

magj2006

Fixed: set initial_prize_contribution to zero after host refunded.

Issue M-3: Missing fee configuration validation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/15

Summary

The `create_tournament` function accepts `FeeConfig` parameters without validating that fee percentages sum to 100% or that individual values fall within acceptable ranges.

Vulnerability Detail

When `create_tournament` is called, the `fee_config` parameter of type `FeeConfig` is passed directly into the `Tournament` struct without any validation. The `FeeConfig` struct contains `fees_to_prize_pool`, `host_fees`, and `war_chest_fees` fields which are expressed in basis points where `10000` represents 100%.

The contract never verifies that these three values sum to exactly 10000, nor does it check that `host_fee_source` and `war_chest_fee_source` contain valid enum values of either `FEE_SOURCE_ENTRY_FEES_ONLY` or `FEE_SOURCE_ENTRY_FEES_AND_INITIAL_PRIZE`.

The `calculate_total_prize_pool` function in `config.move` performs arithmetic using these unvalidated values, and if they do not sum correctly, the resulting prize pool and fee calculations will be incorrect, potentially leaving funds unaccounted for or attempting to distribute more than 100% of collected fees.

Impact

Malformed fee configurations could result in incorrect prize distributions, locked funds, or accounting inconsistencies that affect all participants in the affected tournament.

Code Snippet

Tool Used

Manual Review

Recommendation

Add a `validate_fee_config` function that asserts `fees_to_prize_pool` plus `host_fees` plus `war_chest_fees` equals `10000` and that fee source fields contain valid enum values, then call this validation in `create_tournament` before storing the configuration.

Discussion

magj2006

Fixed: add config::validate_fee_config function to validate fee params.

Issue M-4: Host rewards are locked with no withdrawal mechanism [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/16

Summary

The `tournament_finish` function calculates and stores `host_rewards` but the contract provides no function for hosts to withdraw these earned fees.

Vulnerability Detail

During `tournament_finish` execution, the function calls `config::calculate_total_prize_pool` which returns `host_fees_amount` as one of its outputs, and this value is then stored in `tournament.host_rewards`. The `war_chest_collected` amount is immediately transferred to the treasury address via `token_handler::transfer_fa_from_contract_to_user`, and winners can claim their prizes through `claim_prize`, and hosts can withdraw `host_remaining_initial_prize` through `withdraw_remaining_initial_prize`.

However, a comprehensive review of all public and entry functions in the tournament module reveals no function that reads `tournament.host_rewards` and transfers it to the host. The `host_rewards` field is written but never read for the purpose of disbursement, meaning these funds accumulate in the contract with no programmatic way to extract them.

Impact

All host fees across all tournaments are permanently locked in the contract, resulting in direct financial loss for tournament hosts who expected to receive these fees as compensation.

Code Snippet

Tool Used

Manual Review

Recommendation

Implement a `claim_host_rewards` entry function that validates the caller is the tournament host, checks the tournament is finished, transfers `host_rewards` to the host, and sets `host_rewards` to zero to prevent double claims.

Discussion

magj2006

Fixed: add claim_host_rewards entry function

Issue M-5: Inconsistent status requirements between join and leave functions [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/22

Summary

The `leave_tournament` function requires `TOURNAMENT_STATUS_ONGOING` status while `join_tournament` allows joining during `TOURNAMENT_STATUS_CREATED` and `TOURNAMENT_STATUS_PENDING`, creating a window where users cannot leave. Additionally, it is possible to join even if status is `CANCELLED` or `FINISHED`.

Vulnerability Detail

The `join_tournament` function contains logic that checks if tournament status is `TOURNAMENT_STATUS_CREATED` or `TOURNAMENT_STATUS_PENDING` and transitions it to `TOURNAMENT_STATUS_ONGOING` when `current_time` reaches `tournament.open_time`.

This means users can join a tournament that is in `CREATED` or `PENDING` status before `open_time` transitions it. However, `leave_tournament` has a strict assertion requiring `tournament.status` equals `TOURNAMENT_STATUS_ONGOING` with no allowance for other statuses.

If a user joins during the `CREATED` or `PENDING` phase and later decides to leave before the status transitions to `ONGOING`, they will be unable to do so despite the registration_deadline not having passed. They must wait for another user's `join_tournament` call to trigger the status transition, or for the tournament to become `ONGOING` through some other means.

In the same time, the `join_tournament` totally ignores tournament status - users can join tournaments that are `CANCELLED` or already `FINISHED` as long as `registration_deadline` has not passed.

Impact

Users who join tournaments early cannot leave until the status transitions, reducing user flexibility and potentially trapping entry fees in situations where users change their minds quickly. Additionally, users could join to the cancelled or finished tournaments.

Code Snippet

Tool Used

Manual Review

Recommendation

Modify `leave_tournament` to accept `TOURNAMENT_STATUS_CREATED` and `TOURNAMENT_STATUS_PENDING` in addition to `TOURNAMENT_STATUS_ONGOING`, applying the same status eligibility logic as `join_tournament`, and preventing from joining when the status is `CANCELLED` or `FINISHED`.

Discussion

magj2006

Fixed: add

```
assert!(
            tournament.status == TournamentStatus::CREATED ||
            tournament.status == TournamentStatus::PENDING ||
            tournament.status == TournamentStatus::ONGOING,
            E_INVALID_STATUS
        );
```

to prevent user leave tournament when tournament is other status.

Issue L-1: Permissive blacklist implementation [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/6

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The blacklist check only exists in `join_tournament`, allowing blacklisted users to claim prizes, receive refunds, and leave tournaments without restriction.

Vulnerability Detail

Blacklist validation occurs only when joining new tournaments. Key functions like `claim_prize`, `refund_user`, and `leave_tournament` lack blacklist checks, undermining the sanction system's effectiveness.

Impact

A user blacklisted after joining can still interact normally with the protocol. This might be problematic if user should be completely sanctioned from using the protocol. On the other hand, this form of soft ban may be intentional design of the protocol.

Tool Used

Manual Review

Recommendation

If users should be banned in a hard way, add blacklist check on all user entry functions.

Issue L-2: `max_blacklist_size` is unused [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/7

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `max_blacklist_size` parameter in `ContractConfig` is set but never enforced when adding addresses to the blacklist. On the other hand limiting the blacklist may reduce admins' power to manage problematic users especially in case of sybil attacks.

Impact

No functional impact, but represents dead code and potential confusion about blacklist limitations.

Tool Used

Manual Review

Recommendation

Either implement the size check in `add_to_blacklist` or remove the unused parameter.

Issue L-3: Inefficient loops increase DoS risk [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/8

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Use of linear search in vectors throughout the codebase should be replaced with Table/SmartTable structures to minimize risk of Out Of Gas denial of service. This specifically concerns loops in `add_supported_token`, `remove_supported_token` and in tournament logic - `tournament_finish_entry`, `get_winner_amount` and `calculate_total_winners_amount`.

Vulnerability Detail

The risk of DoS depends on amount of objects in the vector. Iterations over long vectors should be avoided where possible. For instance for supported tokens, table/smart tables can be effectively used to store unique addresses while there is no risk of potential traversing over large arrays.

Impact

Increased DoS risk, dependent on how many elements will be used with those data structures.

Code Snippet

For instance, code for `add_supported_token`:

```
public entry fun add_supported_token(
    resource_account: &signer,
    token_address: address,
) acquires SupportedTokens {
    let resource_addr = signer::address_of(resource_account);
    let supported_tokens = borrow_global_mut<SupportedTokens>(resource_addr);

    // Check if already exists
    let i = 0;
    let len = vector::length(&supported_tokens.tokens);
    while (i < len) {
        let existing_token = *vector::borrow(&supported_tokens.tokens, i);
        assert!(existing_token != token_address, E_INVALID_TOKEN);
        i = i + 1;
```

```
    };

    vector::push_back(&mut supported_tokens.tokens, token_address);
}
```

Tool Used

Manual Review

Recommendation

Refactor the code to use tables/smart tables or ensure limited number of elements will be used with those routines.

Issue L-4: Missing fee validation allows configuration that causes guaranteed underflow [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/9

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Neither the `set_config_entry` or `set_config` functions validate that the sum of fee basis points (`fees_to_prize_pool` + `host_fees`) remains within valid bounds (≤10000).

Vulnerability Detail

he contract calculates `war_chest_percentage` by subtracting fee components from `10000` (representing 100%). When an admin by a mistake sets fee configurations where the sum exceeds `10000` basis points, the calculation attempts to compute a negative value, causing an underflow:

```
// Calculate war chest fees
let war_chest_percentage = 10000 - fee_config.fees_to_prize_pool -
↪  fee_config.host_fees;
```

Impact

Potentially a faulty configuration can be created, temporarily stopping the protocol from working (however this is fully reversible).

Tool Used

Manual Review

Recommendation

Implement a safety check in `set_config_entry` if both those fees do not exceeed `10000` total.

Issue L-5: Tournament can start without host deposit [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/11

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

With strict time settings, it is possible that the tournament starts without initial prize. If that happens, host is unable to deposit it later.

Vulnerability Detail

`join_tournament` automatically transitions a tournament status from `CREATED/PENDING` to `ONGOING` if `current_time >= open_time`. However, `deposit_initial_prize` requires the status to strictly be `PENDING`. If a user joins immediately after `open_time` passes but before the host deposits, the status flips to `ONGOING`, permanently preventing the host from depositing the prize.

Impact

If that happens, the tournament proceeds without the initial prize funds. However this requires a mistake by the host.

Code Snippet

In `smart-contracts/sources/tournament.move`:

```
/// Deposit initial prize (requires admin signature verification)
public entry fun deposit_initial_prize(
    host: &signer,
    tournament_id: u64,
    admin_signature: vector<u8>,
    message: vector<u8>, // Raw message, not hash
) acquires AdminInfo, TournamentStore {
    let host_addr = signer::address_of(host);

    // Verify admin signature using ecrecover-like verification
    let _admin_addr = verify_admin_signature(admin_signature, message);

    let resource_addr = get_resource_account_address();
    let tournament_store = borrow_global_mut<TournamentStore>(resource_addr);
    let tournament = table::borrow_mut(&mut tournament_store.tournaments,
    ↪  tournament_id);
```

```
    assert!(tournament.host == host_addr, E_NOT_HOST);
    assert!(!tournament.is_frozen, E_TOURNAMENT_FROZEN);
    assert!(tournament.status == TOURNAMENT_STATUS_PENDING, E_INVALID_STATUS);
    assert!(tournament.initial_prize_amount > 0, E_INVALID_AMOUNT);
```

Tool Used

Manual Review

Recommendation

Either allow host to add the prize before tournament is closed, or disallow transition to `ON GOING` from status `PENDING` but require host to deposit first, which would change the state to `TOURNAMENT_STATUS_CREATED`.

Issue L-6: Winner addresses not validated as tournament participants [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/17

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `tournament_finish` function accepts arbitrary addresses as winners without verifying they were actual tournament participants.

Vulnerability Detail

When `tournament_finish_entry` is called, it receives `winner_addresses` and `winner_amounts` as separate vectors and combines them into `WinnerInfo` structs that are passed to `tournament_finish`. The `tournament_finish` function then stores these winners in `tournament.winners` and calculates the `total_distributed_prizes`, but at no point does either function check whether each address in `winner_addresses` exists in `tournament.participants`.

This design appears intentional to allow flexibility, but it creates an opportunity for the admin or delegate to designate non-participants as prize winners. The `claim_prize` function only verifies that the caller is in the winners list via `get_winner_amount`, not that they were participants, so designated non-participants could successfully claim prizes.

Impact

Administrators could collude with external parties by designating them as winners despite never having paid entry fees, effectively extracting funds contributed by legitimate participants.

Code Snippet

Tool Used

Manual Review

Recommendation

Add a validation loop in `tournament_finish` that iterates through all winner addresses and asserts each one exists in `tournament.participants` before accepting the winners list.

Issue L-7: Duplicate winner addresses cause unclaimable prizes [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/18

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `tournament_finish_entry` function does not check for duplicate addresses in the winners list, which can result in prize amounts being permanently locked.

Vulnerability Detail

When building the winners vector in `tournament_finish_entry`, the function iterates through `winner_addresses` and creates a `WinnerInfo` struct for each entry without checking if that address already exists in the winners vector being built. If the same address appears twice with different amounts, both entries are stored in `tournament.winners`.

When that user calls `claim_prize`, the `get_winner_amount` function iterates through winners and returns upon finding the first match, so the user receives only the amount from their first entry. After claiming, the `tournament_id` is added to `user_participation.claimed_tournaments`, causing any subsequent claim attempt to fail with `E_ALREADY_CLAIMED`. The prize amount associated with the duplicate entry becomes permanently unclaimable, locked in the contract with no mechanism to recover it.

Impact

Administrative errors in winner list preparation could result in prize funds being permanently locked in the contract, causing financial loss to the intended recipient and breaking expected prize distribution totals.

Code Snippet

Tool Used

Manual Review

Recommendation

Add a nested loop in `tournament_finish_entry` that checks each `winner_addr` against all previously added addresses in the winners vector and aborts with an error if a duplicate is detected.

Issue L-8: Tournament can be finished before registration deadline [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/19

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `tournament_finish` function does not validate timing constraints, allowing tournaments to be concluded before participants have had fair opportunity to compete.

Vulnerability Detail

The `tournament_finish` function checks only that `tournament.status` equals `TOURNAMENT_STATUS_ONGOING` before proceeding to finalize the tournament with the provided winners. It does not compare the current timestamp against `tournament.registration_deadline` or any minimum tournament duration.

A tournament transitions to `TOURNAMENT_STATUS_ONGOING` in `join_tournament` when `current_time` is greater than or equal to `tournament.open_time`, meaning a tournament could theoretically be finished few seconds after becoming `ONGOING`. The config module defines `MIN_TOURNAMENT_DURATION` and `MAX_TOURNAMENT_DURATION` constants, but these are never enforced during `tournament_finish`.

This allows an admin or delegate to prematurely conclude a tournament and designate winners before the competition has meaningfully occurred.

Impact

Premature tournament finalization could be used to manipulate outcomes by ending tournaments before legitimate competition occurs, harming participants who expected a fair duration to compete.

Tool Used

Manual Review

Recommendation

Add a timestamp check in `tournament_finish` that asserts `current_time` is greater than or equal to `tournament.registration_deadline` plus a minimum duration, or track and enforce a `tournament_end_time` field.

Issue L-9: Tournament can be finished below minimum player threshold [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/20

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `tournament_finish` function does not verify that the tournament has met its configured `min_players` requirement before allowing finalization.

Vulnerability Detail

When a tournament is created via `create_tournament`, the `min_players` and `max_players` parameters are validated against config limits and stored in the `Tournament` struct. During `join_tournament`, the function enforces `max_players` by checking that `current_participants` is less than `tournament.max_players` before allowing a new participant.

However, `tournament_finish` performs no corresponding check against `tournament.min_players`. The function reads `tournament.participants` to calculate fee distributions but never asserts that `vector::length` of `tournament.participants` is greater than or equal to `tournament.min_players`.

This allows a tournament to be finished with any number of participants including just one or even zero, despite the `min_players` configuration suggesting a minimum threshold was intended.

Impact

Tournaments can be concluded without meeting previously set minimum participation requirements, potentially violating expectations of hosts who set minimum thresholds and participants who assumed minimum competition levels.

Tool Used

Manual Review

Recommendation

Add an assertion in `tournament_finish` that `vector::length` of `tournament.participants` is greater than or equal to `tournament.min_players` before proceeding with finalization.

Issue L-10: Incorrect error code for tournament ID collision [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-11-matrix-fun-nov-17th/issues/21

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `create_tournament` function uses `E_TOURNAMENT_NOT_FOUND` error code when a tournament already exists, which is incorrect and confusing.

Vulnerability Detail

In `create_tournament`, after borrowing the `TournamentStore`, the function calls `table::contains` to check if the provided `tournament_id` already exists in the tournaments table. The assertion is structured as assert with the negation of `table::contains`, meaning it fails when the tournament DOES exist.

However, the error code used is `E_TOURNAMENT_NOT_FOUND` with value 4, which indicates a tournament was expected but not found. The correct semantic would be something like `E_TOURNAMENT_ALREADY_EXISTS`.

This mismatch between the error condition and the error code makes debugging and error handling more difficult for developers integrating with the contract and for users trying to understand why their transaction failed.

Impact

Developers and users receive misleading error messages when attempting to create a tournament with a duplicate ID, complicating debugging and support processes.

Code Snippet

Tool Used

Manual Review

Recommendation

Define a new error constant `E_TOURNAMENT_ALREADY_EXISTS` and use it for the duplicate `tournament_id` check in `create_tournament`.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.