CHAIN TROOPERS

Zeitgeist

ZRML

Security Assessment Report

June 28th, 2022

Version 1.2

CONFIDENTIAL

Table of Contents

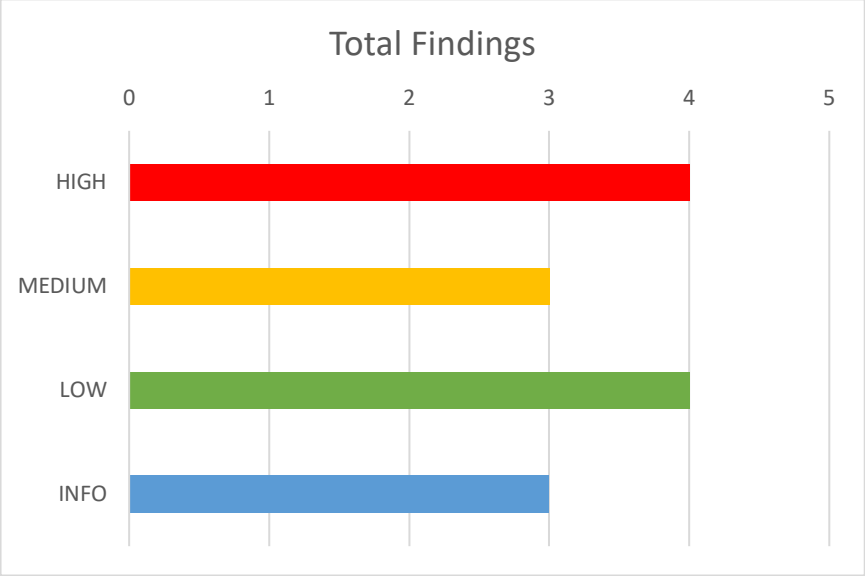
1 Executive Summary

1.1 Introduction

The report contains the results of Zeitgeist ZRML security assessment that took place from April 6th, 2022, to April 28th, 2021. The security engineers performed an in-depth manual analysis of the provided functionalities, and uncovered issues that may be used by adversaries to affect the confidentiality, the integrity, and the availability of the in-scope components.

All the identified vulnerabilities are presented in the report, including their impact and the proposed mitigation strategy, and are ordered by their severity.

In total, the team identified eleven (11) vulnerabilities. There were also three (3) informational issues of no-risk.



All the identified vulnerabilities are presented in the report, including their impact and the proposed mitigation strategy, and are ordered by their severity. A retesting phase was carried out on May 14th, 2022, and June 28th, 2022, and the results are presented in Section 6.

1.2 Assessment Results

The assessment results revealed that the in-scope application components were mainly vulnerable to four (4) Business Logic and Data Validation issues of HIGH risk. Regarding the Business Logic issues, it was identified that the "authorized" pallet was not able to handle cases in which the same user was authorized to resolve disputes in multiple markets *('5.1.1 - [authorized] Lack of support for common authorized user in multiple markets at "authorize_market_outcome" call'*). In these cases, it was not possible for this user to update the outcome of any of the disputed markets except from the last one, leading to a denial-of-service (DoS) vulnerability. Furthermore, another business logic issue was found to affect the "*report*" functionality, which did not take into account the current state of the market (*'5.1.3 - [prediction-markets] Market state is ignored in "Report" function').* Adversaries could exploit this functionality to perform actions that need the market to be in "Active" state, or in order to block the user who proposed the market from canceling it and getting back the deposit, leading again to a DoS vulnerability.

In reference to the High-risk Data Validation issues, the team identified that a user with the approval origin could reject a market that is permissionless *('5.1.2 - [prediction-markets] A permissionless market can be rejected in "reject_market")',* even if this role did not provide a such authorization. As a result, an adversary who had this role could abuse this functionality to escalate their privileges and forcefully reject *permissionless markets,* leading to another DoS attack. Finally, another Data Validation issue was found to affect the "*create_categorical_market*", which was not marked as transactional and was not checking if the calling functions were performed successfully (*'5.1.4 - [prediction-markets] Missing transactional annotation in "create_categorical_market"')*. As a result, in case of error, the legitimate users' assets could remain reserved.

The in-scope components were also affected by three (3) Data Validation and Business Logic vulnerabilities of MEDIUM risk. Regarding the MEDIUM-risk Data Validation issue, it was found that joining a subsidy pool did not require a minimum amount (*'5.2.3 - [swaps] Minimum amount not required in "pool_join_subsidy"'*), allowing adversaries to abuse this functionality and exhaust the available resources. In reference to the MEDIUM-risk Business logic issues, it was found that the reject function did not take into account the state of the market *('5.2.1 - [prediction-markets] Market state ignored in "reject_market" function'),* and

could be used by a user with the Approval Origin even if the market was not in "Proposed" state or users had already acquired the corresponding market shares. Furthermore, in this case, the related outcome market shares would not be handled by the reject functionality (*'5.2.2 - [prediction-markets] Reject market does not manage the related outcome assets').*

There were also four (4) vulnerabilities of LOW risk and three (3) findings of no-risk (INFORMATIONAL). Chaintroopers recommend the immediate mitigation of all HIGH and MEDIUM-risk issues. It is also advisable to address all LOW and INFORMATIONAL findings to enhance the overall security posture of the components.

1.2.1 Retesting Results

Results from retesting carried out on June 2022, determined that all the reported HIGH and MEDIUM risk issues were sufficiently addressed (7 out of 14 findings).

Furthermore, the remaining four (4) LOW risk issues and three (3) INFORMATIONAL issues were also sufficiently mitigated (7 out of 14 findings).

1.3 Summary of Findings

The following findings were identified in the examined source code:

Vulnerability Name	Status	Retest Status	Page
[authorized] Lack of support for common authorized user in multiple markets at "authorize_market_outcome" call	HIGH	CLOSED	15
[prediction-markets] A permissionless market can be rejected in "reject_market"	HIGH	CLOSED	19
[prediction-markets] Market state is ignored in "Report" function	HIGH	CLOSED	23
[prediction-markets] Missing transactional annotation in "create_categorical_market"	HIGH	CLOSED	28
[prediction-markets] Market state ignored in "reject_market" function	MEDIUM	CLOSED	31
[prediction-markets] Reject market does not manage the related outcome assets	MEDIUM	CLOSED	35
[swaps] Minimum amount not required in "pool_join_subsidy"	MEDIUM	CLOSED	39
[prediction-markets] Admin functions without "deposit_event"	LOW	CLOSED	43
[swaps] User defined market type in "admin_set_pool_as_stale" at "swaps"	LOW	CLOSED	47
[prediction-markets] Minimum amount not required in "buy_complete_set"	LOW	CLOSED	51
[swaps] Admin functions without "deposit_event" at "swaps	LOW	CLOSED	55

[prediction-markets] Unlisted dispatch function "deploy_swap_pool_and_additional_liquidity"	INFO	CLOSED	59
[prediction-markets] Market Period Upper Limits not checked	INFO	CLOSED	64
[prediction-markets] Incorrect start range in "admin_move_market_to_closed"	INFO	CLOSED	68

2 Assessment Description

2.1 Target Description

Zeitgeist is an open prediction market platform that is built with Polkadot's blockchain framework, Substrate. It provides the best interface to decentralized prediction market primitives, such as the Categorical, Scalar, and Combinatorial prediction markets, and includes innovations in trading mechanisms such as the Liquidity-Sensitive Logarithmic Market Scoring Rule (LS-LMSR) AMM. The project consists of several Substrate pallets that provide different functionalities.

The "*prediction-market*" pallet is the overarching pallet that is used to create prediction markets, deploy a swap pool for the markets and to handle the market's lifecycle, including an optional early approval phase, a closing, report, dispute, and resolution phase.

The "*swaps*" pallet implements the pool and the automated market maker (AMM) that users trade with. It offers an abstraction over the pools that contain the assets that can be traded against the AMM. It also offers functionality to execute a trade and to provide and remove liquidity.

Once a market is closed and an outcome is reported (which happens within the prediction market pallet), a phase begins during which the reported outcome can be disputed. The prediction market pallet uses the *DisputeApi* that is provided by the authorized and "*simple-disputes*" pallet to handle disputes. During market creation the creator can select one of the available dispute mechanisms.

All the mentioned pallets with exception of the swaps pallet use the "*market-commons pallet*", which serves as a shared storage that abstracts storage manipulations in regard to a market.

2.2 In-Scope Components

The following list specifies the path to crates within the Zeitgeist repository that should be subject to this audit. Tests are excluded.

- *runtime*

- *zrml/authorized*
- *zrml/market-commons*
- *zrml/prediction-markets*
- *zrml/simple-disputes*
- *zrml/swaps*

The "*Rikiddo*" scoring rule is not within the scope of this audit. Only markets that are created with the scoring rule "*ScoringRule::CPMM*" are relevant.

The following functions within the prediction-markets pallet are excluded from the audit:

- *process_subsidy_collecting_markets*
- *start_subsidy*

The following functions within the swaps pallet are excluded from the audit:

- *pool_exit_subsidy*
- *pool_join_subsidy*
- *distribute_pool_share_rewards*
- *destroy_pool_in_subsidy_phase*
- *end_subsidy_phase*

The components are located at the following URL:

https://github.com/zeitgeistpm/zeitgeist/tree/main/zrml

Component	Commit Identifier
ZRML	*fee23e54c8875f1e0542ad101ef74bd96ed97aad*

3 Methodology

3.1 Assessment Methodology

Chaintroopers' methodology attempts to bridge the penetration testing and source code reviewing approaches in order to maximize the effectiveness of a security assessment.

Traditional pentesting or source code review can be done individually and can yield great results, but their effectiveness cannot be compared when both techniques are used in conjunction.

In our approach, the application is stress tested in all viable scenarios though utilizing penetration testing techniques with the intention to uncover as many vulnerabilities as possible. This is further enhanced by reviewing the source code in parallel to optimize this process.

When feasible our testing methodology embraces the Test-Driven Development process where our team develops security tests for faster identification and reproducibility of security vulnerabilities. In addition, this allows for easier understanding and mitigation by development teams.

Chaintroopers' security assessments are aligned with OWASP TOP10 and NIST guidance.

This approach, by bridging penetration testing and code review while bringing the security assessment in a format closer to engineering teams has proven to be highly effective not only in the identification of security vulnerabilities but also in their mitigation and this is what makes Chaintroopers' methodology so unique.

3.2 Smart Contracts

The testing methodology used is based on the empirical study "Defining Smart Contract Defects on Ethereum" by J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo and T. Chen, in IEEE Transactions on Software Engineering, and the security best practices as described in "Security Considerations" section of the solidity wiki.

The following is a non-exhaustive list of security vulnerabilities that are identified by our methodology during the examination of the in-scope contract:

- Unchecked External Calls
- Strict Balance Equality
- Transaction State Dependency
- Hard Code Address
- Nested Call
- Unspecified Compiler Version
- Unused Statement
- Missing Return Statement
- Missing Reminder
- High Gas Consumption Function Type
- DoS Under External Influence
- Unmatched Type Assignment
- Re-entrancy
- Block Info Dependency
- Deprecated APIs
- Misleading Data Location
- Unmatched ERC-20 standard
- Missing Interrupter
- Greedy Contract
- High Gas Consumption Data Type

In Substrate Pallets, the list of vulnerabilities that are identified also includes:

- Static or Erroneously Calculated Weights
- Arithmetic Overflows
- Unvalidated Inputs
- Runtime Panic Conditions
- Missing Storage Deposit Charges
- Non-Transactional Dispatch Functions
- Unhandled Errors &Unclear Return Types
- Missing Origin Authorization Checks

4 Scoring System

4.1 CVSS

All issues identified as a result of Chaintroopers' security assessments are evaluated based on Common Vulnerability Scoring System version 3.1 (https://www.first.org/cvss/).

With the use of CVSS, taking into account a variety of factors a final score is produced ranging from 0 up to 10. The higher the number goes the more critical an issue is.

The following table helps provide a qualitative severity rating:

Rating	CVSS Score
None/Informational	0.0
Low	0.1-3.9
Medium	4.0-6.9
High	7.0-8.9
Critical	9.0-10.0

Issues reported in this document contain a CVSS Score section, this code is provided as an aid to help verify the logic of the team behind the evaluation of a said issue. A CVSS calculator can be found in the following URL:

https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator

5 Identified Findings

5.1 High Severity Findings

5.1.1 [authorized] Lack of support for common authorized user in multiple markets at "authorize_market_outcome" call

Description	HIGH

It was identified that the authorized pallet is not able to handle cases in which a user is authorized in multiple markets. The specific pallet in one of the available arbitration systems for resolving disputes between shareholders and their brokers. It offers authorized resolution of disputes, by allowing the authorized user to provide the outcome of the disputed market.

In the examined implementation, it was found that the user who requests to resolve a dispute using this pallet does not need to provide a market identifier. Instead, the market identifier is located automatically by iterating on the stored records. However, this operation cannot handle cases in which the same user is also authorized to resolve disputes in more than one markets.

More precisely, the issue is located at the "*authorize_market_outcome*" functionality. The market identifier is located using the "*markets.iter().find()*" operation. The first market that is found with that account identifier is taken into consideration for setting the provided "*OutcomeReport*":

```
File: zeitgeist/zrml/authorized/src/lib.rs
45:     #[pallet::call]
46:     impl<T: Config> Pallet<T> {
            ...
50:         pub fn authorize_market_outcome(
                ...
53:          ) -> DispatchResult {
                    ...
56:             let market_id = if let Some(rslt) = markets.iter().find(|el|
{
57:                 if let MarketDisputeMechanism::Authorized(ref account_id)
= el.1.mdm {
```

```
58:                     account_id == &who
59:                 } else {
60:                     false
61:                 }
62:             }) {
63:                 rslt.0
64:             } else {
65:                                                                 return
Err(Error::<T>::AccountIsNotLinkedToAnyAuthorizedMarket.into());
66:             };
67:
68:             Outcomes::<T>::insert(market_id, who, outcome);
69:
```

Impact

In case that a legitimate user is authorized in more than one markets, this issue will not allow them from inserting the provided outcome in any of the previously created markets, leading to a denial-of-service (DoS) vulnerability.

Only the last inserted market can be updated. Furthermore, the result of the function will not provide any feedback to the user regarding the market that was updated, as there is no emitted event. The presented issue can be replicated with the following test case:

```
#[test]
fn authorize_market_outcome_inserts_a_new_outcome_when_two_markets() {
    ExtBuilder::default().build().execute_with(|| {
        Markets::<Runtime>::insert(0, market_mock::<Runtime>(ALICE));
        Markets::<Runtime>::insert(1, market_mock::<Runtime>(ALICE));
        Authorized::authorize_market_outcome(Origin::signed(ALICE),
OutcomeReport::Categorical(1))
            .unwrap();
        assert_eq!(Outcomes::<Runtime>::get(1,           ALICE).unwrap(),
OutcomeReport::Categorical(1));
    });
}
```

Finally, if the service allows users to select arbitrary users as authorized users for a market on demand, an adversary could exploit this issue in order to set a victim user as authorized to a custom attacker-controlled market, in order to prevent him from inserting an outcome to their legitimate market.

```
#[test]
fn
authorize_market_outcome_inserts_a_new_outcome_when_two_markets_forced()
{
    ExtBuilder::default().build().execute_with(|| {
        Markets::<Runtime>::insert(0, market_mock::<Runtime>(ALICE));
        Markets::<Runtime>::insert(1, market_mock::<Runtime>(BOB));
        Markets::<Runtime>::mutate(1, |el| {
            el.as_mut().unwrap().mdm                                    =
MarketDisputeMechanism::Authorized(ALICE);
        });
        Authorized::authorize_market_outcome(Origin::signed(ALICE),
OutcomeReport::Categorical(1))
            .unwrap();
        assert_eq!(Outcomes::<Runtime>::get(1,           ALICE).unwrap(),
OutcomeReport::Categorical(1));
    });
}
```

Recommendation

It is recommended to add support for having a common authorized user between multiple markets.

The user should be able to select the market by providing the market identifier as a parameter to the “authorize_market_outcome“ functionality.

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.1.2 [prediction-markets] A permissionless market can be rejected in "reject_market"

Description	HIGH

It was identified that the user with the "*Approval Origin*" can reject a market that is permissionless. The "Approval Origin" is a custom origin that can be used to perform authorization checks inside functions of specific modules in the runtime. In the specific case, the "Approval Origin" is a privilege of the advisory committee. The committee is the on-chain governing body of Zeitgeist that is responsible for maintaining a list of high-quality markets and slash low quality markets.

Normally, the "*reject_market*" function is designed to reject a "*Proposed*" market of "Advised" type that is waiting for approval from the advisory committee. However, it was found that no check is performed to validate the market type. As a result, other market types can also be rejected using this functionality.

The issue is located at the following code:

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
906:
907:          /// Rejects a market that is waiting for approval from the
advisory committee.
908:         #[pallet::weight(T::WeightInfo::reject_market())]
909:              pub fn reject_market(origin: OriginFor<T>, market_id:
MarketIdOf<T>) -> DispatchResult {
910:             T::ApprovalOrigin::ensure_origin(origin)?;
911:
912:             let market = T::MarketCommons::market(&market_id)?;
913:             let creator = market.creator;
914:           let (imbalance, _) = CurrencyOf::<T>::slash_reserved_named(
915:                 &RESERVE_ID,
916:                 &creator,
917:                 T::AdvisoryBond::get(),
918:             );
919:             // Slashes the imbalance.
920:             T::Slash::on_unbalanced(imbalance);
921:             T::MarketCommons::remove_market(&market_id)?;
```

```
922:             Self::deposit_event(Event::MarketRejected(market_id));
923:             Self::deposit_event(Event::MarketDestroyed(market_id));
924:             Ok(())
925:         }
```

For example, the following test case can be used to create a Permissionless market, change its status to “Reported” instead of “Proposed”, and then use the “*reject_market*” function to remove it from the list:

```
#[test]
fn it_allows_the_advisory_origin_to_reject_permissionless_markets() {
    ExtBuilder::default().build().execute_with(|| {
        // Creates an advised market.
        assert_ok!(PredictionMarkets::create_categorical_market(
            Origin::signed(ALICE),
            BOB,
            MarketPeriod::Block(0..100),
            gen_metadata(2),
            MarketCreation::Permissionless,
            3,
            MarketDisputeMechanism::SimpleDisputes,
            ScoringRule::CPMM
        ));

        deploy_swap_pool(MarketCommons::market(&0).unwrap(), 0).unwrap();

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(ALICE), 0,
1 * BASE));

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(BOB), 0, 2
* BASE));

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(CHARLIE),
0, 3 * BASE));

        run_to_block(100);
        assert_ok!(PredictionMarkets::report(
            Origin::signed(BOB),
```

```
            0,
            OutcomeReport::Categorical(2)
        ));
        // make sure it's in status reported and not proposed
        let market = MarketCommons::market(&0);
        assert_eq!(market.unwrap().status, MarketStatus::Reported);

        // Now it should work from SUDO
        assert_ok!(PredictionMarkets::reject_market(Origin::signed(SUDO),
0));

        assert_noop!(
            MarketCommons::market(&0),
            zrml_market_commons::Error::<Runtime>::MarketDoesNotExist
        );
    });
}
```

However, the user with the "*Approval Origin*" does not necessarily need to be also authorized to reject an active permissionless market. As a result, this action could be unauthorized.

Impact

An adversary who had this role could abuse this functionality to escalate their privileges and forcefully reject *permissionless markets* of other legitimate users, leading to a Denial of Service (DoS) attack.

Recommendation

It is recommended to validate if the market type is "Advised". For example:

```
ensure!(m.creation== MarketCreation::Advised,
Error::<T>::MarketIsNotAdvised);
```

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.1.3 [prediction-markets] Market state is ignored in "Report" function

Description	HIGH

It was identified that the "*report*" functionality does not take into account the state of the market (e.g., if the market is still in Proposed state). Prediction markets are speculative markets that trade on future outcomes. While a prediction market is active, traders can permissionlessly trade on outcome assets. However, when the market has reached the end of its period, selected users are able to provide the final outcome of the market.

The report functionality is designed for this purpose; to allow users to set the outcome of a market. However, it was found that there is no check to verify if the market was "Active":

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
927:         /// Reports the outcome of a market.
928:         ///
929:         #[pallet::weight(T::WeightInfo::report())]
930:         #[transactional]
931:         pub fn report(
932:             origin: OriginFor<T>,
933:             market_id: MarketIdOf<T>,
934:             outcome: OutcomeReport,
935:         ) -> DispatchResult {
936:             let sender = ensure_signed(origin.clone())?;
937:
938:                                                 let    current_block    =
<frame_system::Pallet<T>>::block_number();
939:                 let market_report = Report { at: current_block, by:
sender.clone(), outcome };
940:
941:             T::MarketCommons::mutate_market(&market_id, |market| {
942:                 // TODO make this a conditional check
943:                                 //  ensure!(outcome  <=  market.outcomes(),
Error::<T>::OutcomeOutOfRange);
944:                                         ensure!(market.report.is_none(),
Error::<T>::MarketAlreadyReported);
945:
```

```
946:                 Self::ensure_market_is_closed(&market.period)?;
947:
948:                 let mut should_check_origin = false;
949:                 match market.period {
950:                     MarketPeriod::Block(ref range) => {
951:                                     if current_block <= range.end +
T::ReportingPeriod::get().into() {
952:                             should_check_origin = true;
953:                         }
954:                     }
955:                     MarketPeriod::Timestamp(ref range) => {
956:                                         let rp_moment: MomentOf<T> =
T::ReportingPeriod::get().into();
957:                             let reporting_period_in_ms = rp_moment *
MILLISECS_PER_BLOCK.into();
958:                             if T::MarketCommons::now() <= range.end +
reporting_period_in_ms {
959:                             should_check_origin = true;
960:                         }
961:                     }
962:                 }
963:
964:                 if should_check_origin {
965:                     let sender_is_oracle = sender == market.oracle;
966:                                         let origin_has_permission =
T::ResolveOrigin::ensure_origin(origin).is_ok();
967:                     ensure!(
968:                         sender_is_oracle || origin_has_permission,
969:                         Error::<T>::ReporterNotOracle
970:                     );
971:                 }
972:
973:                 market.report = Some(market_report.clone());
974:                 market.status = MarketStatus::Reported;
975:
976:                 Ok(())
977:             })?;
978:
979:             MarketIdsPerReportBlock::<T>::try_mutate(&current_block,
|ids| {
```

```
980:                                     ids.try_push(market_id).map_err(|_|
<Error<T>>::StorageOverflow)
981:             })?;
982:
983:             Self::deposit_event(Event::MarketReported(
984:                 market_id,
985:                 MarketStatus::Reported,
986:                 market_report,
987:             ));
988:             Ok(())
989:         }
```

For example, an “Advised market” can be reported even if it was never approved by the Advisory committee.

Impact

An adversary can abuse this issue in order to circumvent actions that need the market to be in Active state, or in order to block the user who proposed the market from canceling it and getting back the deposit.

For example, in case that an Advised market is created, the user who created the market (ALICE), can cancel the market, and return the deposit.

```
#[test]
fn check_proposed_canceled() {
    ExtBuilder::default().build().execute_with(|| {
        simple_create_categorical_market::<Runtime>(
            MarketCreation::Advised,
            0..1,
            ScoringRule::CPMM,
        );
        run_to_block(1);
        let market = MarketCommons::market(&0);
        assert_eq!(market.unwrap().status, MarketStatus::Proposed);
        let market = MarketCommons::market(&0).unwrap();
        assert_eq!(market.report.is_none(), true);

assert_ok!(PredictionMarkets::cancel_pending_market(Origin::signed(ALICE)
, 0));
    });
```

```
}
```

However, if BOB reports the proposed market, then the state is changed and canceling is impossible.

```
#[test]
fn check_proposed_reported_canceled() {
    ExtBuilder::default().build().execute_with(|| {
        simple_create_categorical_market::<Runtime>(
            MarketCreation::Advised,
            0..1,
            ScoringRule::CPMM,
        );
        run_to_block(1);
        let market = MarketCommons::market(&0);
        assert_eq!(market.unwrap().status, MarketStatus::Proposed);
        assert_ok!(PredictionMarkets::report(
            Origin::signed(BOB),
            0,
            OutcomeReport::Categorical(1)
        ));

assert_ok!(PredictionMarkets::cancel_pending_market(Origin::signed(ALICE)
, 0));
    });
}
```

This will fail because Bob took advantage of the report function and changed the state to reported, making canceling impossible.

```
---- tests::check_proposed_reported_canceled stdout ----
thread 'tests::check_proposed_reported_canceled' panicked at 'Expected
Ok(_). Got Err(
    Other(
        "Market must be pending approval.",
    ),
```

Recommendation

It is recommended to validate if the market type is not in “Proposed” state. For example:

```
ensure!(m.status == MarketStatus:: Active, Error::<T>::
MarketIsNotActive);
```

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.1.4 [prediction-markets] Missing transactional annotation in "create_categorical_market"

Description	HIGH

It was identified that the publicly available dispatch call which is provided for creating categorical markets is not marked as transactional. Substrate uses Rust macros to aggregate the logic derived from pallets that are implemented for a runtime. The transactional macro can be used to execute the annotated runtime function in a new storage transaction. It provides a convenient way to make runtime functions atomic, as all changes to storage performed by the annotated function will be discarded if it returns an error, or committed otherwise.

The issue exists at the "create_categorical_market" functionality:

File: zeitgeist/zrml/prediction-markets/src/lib.rs

```
377:         #[pallet::weight(T::WeightInfo::create_categorical_market())]
378:         pub fn create_categorical_market(
379:             origin: OriginFor<T>,
380:             oracle: T::AccountId,
381:             period: MarketPeriod<T::BlockNumber, MomentOf<T>>,
382:             metadata: MultiHash,
383:             creation: MarketCreation,
384:             categories: u16,
385:             mdm: MarketDisputeMechanism<T::AccountId>,
386:             scoring_rule: ScoringRule,
387:         ) -> DispatchResultWithPostInfo {
388:             let sender = ensure_signed(origin)?;
389:             Self::ensure_market_is_active(&period)?;
390:
391:                         ensure!(categories >= T::MinCategories::get(),
<Error<T>>::NotEnoughCategories);
392:                         ensure!(categories <= T::MaxCategories::get(),
<Error<T>>::TooManyCategories);
393:
394:             if scoring_rule == ScoringRule::RikiddoSigmoidFeeMarketEma
{
395:                 Self::ensure_market_start_is_in_time(&period)?;
396:             }
```

```
397:
398:             // Require sha3-384 as multihash.
399:             let MultiHash::Sha3_384(multihash) = metadata;
400:               ensure!(multihash[0] == 0x15 && multihash[1] == 0x30,
<Error<T>>::InvalidMultihash);
401:
402:             let status: MarketStatus = match creation {
403:                 MarketCreation::Permissionless => {
404:                         let required_bond = T::ValidityBond::get() +
T::OracleBond::get();
405:                           CurrencyOf::<T>::reserve_named(&RESERVE_ID,
&sender, required_bond)?;
406:
407:                     if scoring_rule == ScoringRule::CPMM {
408:                         MarketStatus::Active
409:                     } else {
410:                         MarketStatus::CollectingSubsidy
411:                     }
412:                 }
413:                 MarketCreation::Advised => {
414:                         let required_bond = T::AdvisoryBond::get() +
T::OracleBond::get();
415:                           CurrencyOf::<T>::reserve_named(&RESERVE_ID,
&sender, required_bond)?;
416:                     MarketStatus::Proposed
417:                 }
418:             };
419:
420:             let market = Market {
421:                 creation,
422:                 creator_fee: 0,
423:                 creator: sender,
424:                 market_type: MarketType::Categorical(categories),
425:                 mdm,
426:                 metadata: Vec::from(multihash),
427:                 oracle,
428:                 period,
429:                 report: None,
430:                 resolved_outcome: None,
431:                 scoring_rule,
432:                 status,
```

```
433:             };
434:                                                 let    market_id    =
T::MarketCommons::push_market(market.clone())?;
435:             let mut extra_weight = 0;
436:
437:             if market.status == MarketStatus::CollectingSubsidy {
438:              extra_weight = Self::start_subsidy(&market, market_id)?;
439:             }
440:
441:                  Self::deposit_event(Event::MarketCreated(market_id,
market));
442:
443:
Ok(Some(T::WeightInfo::create_categorical_market().saturating_add(extra_w
eight)).into())
444:         }
445:
```

Impact

In the specific case, if an error occurs, the legitimate users' assets could remain reserved.

Recommendation

It is advisable to add the "transactional" macro label:

```
#[transactional]
```

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2 Medium Severity Findings

5.2.1 [prediction-markets] Market state ignored in "reject_market" function

Description	MEDIUM

It was identified that the reject function does not take into account the state of the market (e.g., if the market is not in Proposed state).

Normally, the *"reject_market"* function is designed to reject a "*Proposed*" market of "Advised" type that is waiting for approval from the advisory committee. However, it was found that no check is performed to validate the market status.

As a result, a market in "Reported" state can also be rejected using this functionality.

File: zeitgeist/zrml/prediction-markets/src/lib.rs

```
907:         /// Rejects a market that is waiting for approval from the
advisory committee.
908:         #[pallet::weight(T::WeightInfo::reject_market())]
909:             pub fn reject_market(origin: OriginFor<T>, market_id:
MarketIdOf<T>) -> DispatchResult {
910:             T::ApprovalOrigin::ensure_origin(origin)?;
911:
912:             let market = T::MarketCommons::market(&market_id)?;
913:             let creator = market.creator;
914:                                         let    (imbalance,    _)    =
CurrencyOf::<T>::slash_reserved_named(
915:                 &RESERVE_ID,
916:                 &creator,
917:                 T::AdvisoryBond::get(),
918:             );
919:             // Slashes the imbalance.
920:             T::Slash::on_unbalanced(imbalance);
921:             T::MarketCommons::remove_market(&market_id)?;
922:             Self::deposit_event(Event::MarketRejected(market_id));
923:             Self::deposit_event(Event::MarketDestroyed(market_id));
924:             Ok(())
```

```
925:         }
```

On the contrary, the "*approve_market*" does check if the market is still in Proposed state.

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
250:         #[pallet::weight(T::WeightInfo::approve_market())]
251:         pub fn approve_market(
252:             origin: OriginFor<T>,
253:             market_id: MarketIdOf<T>,
254:         ) -> DispatchResultWithPostInfo {
255:             T::ApprovalOrigin::ensure_origin(origin)?;
 ...
259:             T::MarketCommons::mutate_market(&market_id, |m| {
260:                         ensure!(m.status == MarketStatus::Proposed,
Error::<T>::MarketIsNotProposed);
   ...
272:             })?;
 ...
276:         }
277:
```

A user with the ApprovalOrigin role can accidentally reject an already approved market, even if there are already acquired market shares by other legitimate users:

```
#[test]
fn reject_approved_market() {
    ExtBuilder::default().build().execute_with(|| {
        assert_ok!(PredictionMarkets::create_categorical_market(
            Origin::signed(ALICE),
            BOB,
            MarketPeriod::Block(0..100),
            gen_metadata(2),
            MarketCreation::Advised,
            3,
```

```
            MarketDisputeMechanism::SimpleDisputes,
            ScoringRule::CPMM
        ));

assert_ok!(PredictionMarkets::approve_market(Origin::signed(SUDO), 0));
        deploy_swap_pool(MarketCommons::market(&0).unwrap(),
0).unwrap();

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(ALICE),
0, 1 * BASE));

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(BOB), 0, 2
* BASE));

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(CHARLIE),
0, 3 * BASE));
        run_to_block(50);

assert_ok!(PredictionMarkets::reject_market(Origin::signed(SUDO), 0));
    });
}
```

Impact

An adversary who had this role could abuse this functionality to forcefully reject an advised market which is already in “Reported” phase, leading to a Denial of Service (DoS) attack.

Recommendation

It is recommended to validate if the market state is in “Proposed” state. For example:

```
ensure!(m.status == MarketStatus::Proposed,
Error::<T>::MarketIsNotProposed);
```

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.2 [prediction-markets] Reject market does not manage the related outcome assets

Description	MEDIUM

It was identified that when the "reject_market" is used, the related outcome assets are not removed. Normally, the *"reject_market"* function is designed to reject a "*Proposed*" market that is waiting for approval from the advisory committee. Advised markets that are in Proposed state shouldn't have any related outcome assets. However, as it was identified in issues 5.1.2, 5.1.3 and 5.2.1 this is possible.

File: zeitgeist/zrml/prediction-markets/src/lib.rs

```
907:          /// Rejects a market that is waiting for approval from the
advisory committee.
908:         #[pallet::weight(T::WeightInfo::reject_market())]
909:              pub fn reject_market(origin: OriginFor<T>, market_id:
MarketIdOf<T>) -> DispatchResult {
910:             T::ApprovalOrigin::ensure_origin(origin)?;
911:
912:             let market = T::MarketCommons::market(&market_id)?;
913:             let creator = market.creator;
914:           let (imbalance, _) = CurrencyOf::<T>::slash_reserved_named(
915:                 &RESERVE_ID,
916:                 &creator,
917:                 T::AdvisoryBond::get(),
918:             );
919:             // Slashes the imbalance.
920:             T::Slash::on_unbalanced(imbalance);
921:             T::MarketCommons::remove_market(&market_id)?;
922:             Self::deposit_event(Event::MarketRejected(market_id));
923:             Self::deposit_event(Event::MarketDestroyed(market_id));
924:             Ok(())
925:         }
926:
```

On the contrary, other functions that remove markets (such as "*admin_destroy_market()*") will also remove the outcome assets.

File: zeitgeist/zrml/prediction-markets/src/lib.rs

```
134:         pub fn admin_destroy_market(
135:             origin: OriginFor<T>,
136:             market_id: MarketIdOf<T>,
137:         ) -> DispatchResultWithPostInfo {
138:             T::DestroyOrigin::ensure_origin(origin)?;
139:
                 ...
149:
150:             let mut outcome_assets_iter = outcome_assets.into_iter();
151:
152:             // Delete of this market's outcome assets.
153:             let mut manage_outcome_asset = |asset: Asset<_>| -> usize
{
154:                                 let (total_accounts, accounts) =
T::Shares::accounts_by_currency_id(asset);
155:                                                         share_accounts =
share_accounts.saturating_add(accounts.len());
156:                 T::Shares::destroy_all(asset, accounts.iter().cloned());
157:                     total_accounts
158:             };
159:
                ...
185:         }
186:
```

For example, the following test is using the function "r*eject_market*" to remove a market that had already several associated assets. Then the outcome assets are checked and verified.

```
#[test]
fn reject_doesnot_manage_outcome_assets() {
    ExtBuilder::default().build().execute_with(|| {
        assert_ok!(PredictionMarkets::create_categorical_market(
            Origin::signed(ALICE),
```

```
            BOB,
            MarketPeriod::Block(0..100),
            gen_metadata(2),
            MarketCreation::Advised,
            3,
            MarketDisputeMechanism::SimpleDisputes,
            ScoringRule::CPMM
        ));

assert_ok!(PredictionMarkets::approve_market(Origin::signed(SUDO), 0));
        deploy_swap_pool(MarketCommons::market(&0).unwrap(), 0).unwrap();

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(ALICE),  0,
1 * BASE));

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(BOB), 0, 2 *
BASE));

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(CHARLIE),
0, 3 * BASE));
        run_to_block(50);
        let market = MarketCommons::market(&0).unwrap();
        assert_ok!(PredictionMarkets::reject_market(Origin::signed(SUDO),
0));
        let assets = PredictionMarkets::outcome_assets(0, &market);
        for asset in assets.iter() {
            let bal = Tokens::free_balance(*asset, &CHARLIE);
            assert_eq!(bal, 3 * BASE);
        }
```

However, if we replace "reject_market" with "admin_destroy_market", the test will correctly fail, since they will have been correctly removed.

```
---- tests::reject_doesnot_manage_outcome_assets stdout ----
thread    'tests::reject_doesnot_manage_outcome_assets'    panicked    at
'assertion failed: `(left == right)`
  left: `0`,
 right: `30000000000`', zrml/prediction-markets/src/tests.rs:117:13
```

```
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
failures:
    tests::reject_doesnot_manage_outcome_assets
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 28 filtered
out; finished in 0.02s
```

Impact

In case that the “reject_market” functionality is used in a market that has already associated outcome assets; the reserved funds will not be released.

Recommendation

It is advisable to also remove any related outcome assets, in a similar way to “*admin_destroy_market*”.

It should be noted that this issue can be fixed by mitigating 5.1.2, 5.1.3 and 5.2.1.

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.3 [swaps] Minimum amount not required in "pool_join_subsidy"

Description	MEDIUM

It was identified that no minimum amount is required in the publicly available dispatch functionality which is provided to join a subsidy pool. In general, the created pools allow the liquidity providers to deposit full outcome shares and earn fees. In the specific case, the functionality is used to add a subsidy to a pool that uses the Rikiddo scoring rule. The provided parameter is the amount of the base currency that would be reserved to be added as subsidy on pool activation. However, it was found that this amount can be zero.

The issue exists at the "pool_join_subsidy " function:

File: /zeitgeist/zrml/swaps/src/lib.rs

```
387:         /// Pool - Add subsidy to a pool that uses the Rikiddo scoring
rule.
388:         ///
389:          /// Reserves `pool_amount` of the base currency to be added
as subsidy on pool activation.
390:         ///
391:         /// # Arguments
392:         ///
393:          /// * `origin`: Liquidity Provider (LP). The account whose
assets should be reserved.
394:         /// * `pool_id`: Unique pool identifier.
395:          /// * `amount`: The amount of base currency that should be
added to subsidy.
396:         #[pallet::weight(T::WeightInfo::pool_join_subsidy())]
397:         pub fn pool_join_subsidy(
398:             origin: OriginFor<T>,
399:             pool_id: PoolId,
400:             amount: BalanceOf<T>,
401:         ) -> DispatchResult {
402:             let who = ensure_signed(origin)?;
403:
404:             <Pools<T>>::try_mutate(pool_id, |pool_opt| {
405:                                                     let    pool    =
pool_opt.as_mut().ok_or(Error::<T>::PoolDoesNotExist)?;
```

```
406:
407:                 ensure!(
408:                                                     pool.scoring_rule  ==
ScoringRule::RikiddoSigmoidFeeMarketEma,
409:                     Error::<T>::InvalidScoringRule
410:                 );
411:                                                   let  base_asset  =
pool.base_asset.ok_or(Error::<T>::BaseAssetNotFound)?;
412:                 T::Shares::reserve(base_asset, &who, amount)?;
413:
414:                                                 let  total_subsidy  =
pool.total_subsidy.ok_or(Error::<T>::PoolMissingSubsidy)?;
415:                 let _ = <SubsidyProviders<T>>::mutate(&pool_id, &who,
|user_subsidy| {
416:                     if let Some(prev_val) = user_subsidy {
417:                         *prev_val += amount;
418:                     } else {
419:                         *user_subsidy = Some(amount);
420:                     }
421:
422:                    pool.total_subsidy = Some(total_subsidy + amount);
423:                 });
424:
425:
Self::deposit_event(Event::<T>::PoolJoinSubsidy(PoolAssetEvent {
426:                     asset: base_asset,
427:                     bound: amount,
428:                     cpep: CommonPoolEventParams { pool_id, who },
429:                     transferred: amount,
430:                 }));
431:
432:                 Ok(())
433:             })
434:         }
```

For example, in the following case, Alice and Charlie are added a pool that uses the Rikiddo scoring rule and reserves zero (0) of the base currency to be added

as subsidy on pool activation. This will lead to just extra inserts that would need to be processed.

```
fn charlie_signed() -> Origin {
    Origin::signed(CHARLIE)
}
#[test]
fn test_subsidy_provider() {
    ExtBuilder::default().build().execute_with(|| {

create_initial_pool_with_funds_for_alice(ScoringRule::RikiddoSigmoidFeeMa
rketEma, false);
        let pool_id = 0;
        // Reserve some funds for subsidy
        assert_ok!(Swaps::pool_join_subsidy(charlie_signed(),    pool_id,
0));
        assert_eq!(Currencies::reserved_balance(ASSET_D, &CHARLIE), 0);
        assert!(crate::SubsidyProviders::<Runtime>::contains_key(pool_id,
CHARLIE));
    });
}
```

Impact

An adversary can abuse this issue in order to create empty entries in the liquidity pools, that will not reserve any amount of assets. These entries can lead to resource exhaustion or being used to create multiple misleading deposit events.

Recommendation

It is advisable to ensure that the provided amount in the "pool_join_subsidy " function is bigger than zero.

```
ensure!(amount > 0, Error::<T>:: NotEnoughAssets);
```

CVSS Score
AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3 Low Severity Findings

5.3.1 [prediction-markets] Admin functions without "deposit_event"

Description	LOW

It was found that certain admin functions do not utilize the "deposit_event". The corresponding events are also missing. A pallet can emit events when it wants to notify external entities about changes or conditions in the runtime to external entities like users, chain explorers, or dApps.

The available events are the following:

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
1239:     #[pallet::event]
1240:     #[pallet::generate_deposit(fn deposit_event)]
1241:     pub enum Event<T>
1242:     where
1243:         T: Config,
1244:     {
1245:             /// Custom addition block initialization logic wasn't
successful
1246:         BadOnInitialize,
1247:           /// A complete set of assets has been bought \[market_id,
amount_per_asset, buyer\]
1248:               BoughtCompleteSet(MarketIdOf<T>, BalanceOf<T>, <T as
frame_system::Config>::AccountId),
1249:       /// A market has been approved \[market_id, new_market_status\]
1250:         MarketApproved(MarketIdOf<T>, MarketStatus),
1251:         /// A market has been created \[market_id, creator\]
1252:                   MarketCreated(MarketIdOf<T>, Market<T::AccountId,
T::BlockNumber, MomentOf<T>>),
1253:         /// A market has been created \[market_id, creator\]
1254:         MarketDestroyed(MarketIdOf<T>),
1255:           /// A market was started after gathering enough subsidy.
\[market_id, new_market_status\]
1256:         MarketStartedWithSubsidy(MarketIdOf<T>, MarketStatus),
1257:           /// A market was discarded after failing to gather enough
subsidy. \[market_id, new_market_status\]
```

```
1258:         MarketInsufficientSubsidy(MarketIdOf<T>, MarketStatus),
1259:         /// A pending market has been cancelled. \[market_id\]
1260:         MarketCancelled(MarketIdOf<T>),
1261:        /// A market has been disputed \[market_id, new_market_status,
new_outcome\]
1262:                        MarketDisputed(MarketIdOf<T>,  MarketStatus,
MarketDispute<T::AccountId, T::BlockNumber>),
1263:              /// A pending market has been rejected as invalid.
\[market_id\]
1264:         MarketRejected(MarketIdOf<T>),
1265:                /// A market has been reported on \[market_id,
new_market_status, reported_outcome\]
1266:                        MarketReported(MarketIdOf<T>,  MarketStatus,
Report<T::AccountId, T::BlockNumber>),
1267:        /// A market has been resolved \[market_id, new_market_status,
real_outcome\]
1268:         MarketResolved(MarketIdOf<T>, MarketStatus, OutcomeReport),
1269:           /// A complete set of assets has been sold \[market_id,
amount_per_asset, seller\]
1270:                SoldCompleteSet(MarketIdOf<T>, BalanceOf<T>, <T as
frame_system::Config>::AccountId),
1271:     }
```

However, certain functions such as the "admin_move_market_to_closed" do not send the event

File: /zeitgeist/zrml/prediction-markets/src/lib.rs

```
187:           /// Allows the `CloseOrigin` to immediately move an open
market to closed.
188:         //
189:         // ***** IMPORTANT *****
190:         //
191:          // Within the same block, operations that interact with the
activeness of the same
192:         // market will behave differently before and after this call.
193:
#[pallet::weight(T::WeightInfo::admin_move_market_to_closed())]
194:         pub fn admin_move_market_to_closed(
195:             origin: OriginFor<T>,
```

```
196:             market_id: MarketIdOf<T>,
197:         ) -> DispatchResult {
198:             T::CloseOrigin::ensure_origin(origin)?;
199:             T::MarketCommons::mutate_market(&market_id, |m| {
200:                 m.period = match m.period {
201:                     MarketPeriod::Block(ref range) => {
202:                                                 let current_block =
<frame_system::Pallet<T>>::block_number();
203:                      MarketPeriod::Block(range.start..current_block)
204:                     }
205:                     MarketPeriod::Timestamp(ref range) => {
206:                         let now = T::MarketCommons::now();
207:                         MarketPeriod::Timestamp(range.start..now)
208:                     }
209:                 };
210:                 Ok(())
211:             })?;
212:             Ok(())
213:         }
214:
```

The same issue also affects the "admin_move_market_to_resolved"

Impact

Events are necessary to notify the off-chain world of successful state transitions. Administration functionalities should emit the corresponding events throughout the system's life cycle in order to provide credibility and confidence in the system.

Recommendation

It is recommended to emit an event related to this administrative functionality. For example:

```
Self::deposit_event(Event::MarketResolved(marketid, resolved_outcome));
Self::deposit_event(Event::MarketClosed(marketid));
```

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.2 [swaps] User defined market type in "admin_set_pool_as_stale" at "swaps"

Description	LOW

It was identified that the market type is provided at "admin_set_pool_as_stale" and at "set_pool_as_stale" functionalities. However, this information could be stored in the pool entity in order to avoid later taking security decisions on user provided parameters.

Initially, the root user calls the "admin_set_pool_as_stale" function with the market type, the pool id and the outcome report as parameters:

File: zeitgeist/zrml/swaps/src/lib.rs

```
085:     #[pallet::call]
086:     impl<T: Config> Pallet<T> {
087:         #[pallet::weight(T::WeightInfo::admin_set_pool_as_stale())]
088:         #[frame_support::transactional]
089:         pub fn admin_set_pool_as_stale(
090:             origin: OriginFor<T>,
091:             market_type: MarketType,
092:             pool_id: PoolId,
093:             outcome_report: OutcomeReport,
094:         ) -> DispatchResult {
095:             ensure_root(origin)?;
096:             Self::set_pool_as_stale(
097:                 &market_type,
098:                 pool_id,
099:                 &outcome_report,
100:                 &Self::pool_account_id(pool_id),
101:             )?;
102:             Ok(())
103:         }
```

Then, the "admin_set_pool_as_stale" retrieves the pool account id and calls the "set_pool_as_stale" function:

File: /zeitgeist/zrml/swaps/src/lib.rs

```
1618:         #[frame_support::transactional]
1619:         fn set_pool_as_stale(
1620:             market_type: &MarketType,
1621:             pool_id: PoolId,
1622:             outcome_report: &OutcomeReport,
1623:             winner_payout_account: &T::AccountId,
1624:         ) -> Result<Weight, DispatchError> {
1625:             let mut extra_weight = 0;
1626:             let mut total_assets = 0;
1627:
1628:             Self::mutate_pool(pool_id, |pool| {
1629:                 if pool.pool_status == PoolStatus::Stale {
1630:                     return Ok(());
1631:                 }
1632:
1633:                     ensure!(pool.pool_status == PoolStatus::Active,
Error::<T>::InvalidStateTransition);
1634:                 let base_asset = pool.base_asset;
1635:                 let mut winning_asset: Result<_, DispatchError> =
1636:                     Err(Error::<T>::WinningAssetNotFound.into());
1637:
1638:                 if let MarketType::Categorical(_) = market_type {
1639:                                         let  base_asset_or_default  =
base_asset.unwrap_or(Asset::Ztg);
1640:
1641:                 if let OutcomeReport::Categorical(winning_asset_idx)
= outcome_report {
1642:                         pool.assets.retain(|el| {
1643:                             if let Asset::CategoricalOutcome(_, idx)
= *el {
1644:                                 if idx == *winning_asset_idx {
1645:                                     winning_asset = Ok(*el);
1646:                                     return true;
1647:                                 };
1648:                             }
1649:
1650:                             *el == base_asset_or_default
1651:                         });
1652:                     }
```

```
1653:
1654:                     total_assets = pool.assets.len();
1655:                 }
                     ...
```

In general, the same function could be called with an incorrectly defined market type, which however wouldn't be handled by the "set_pool_as_stale" with an expected error condition:

```
#[test]
fn call_admin_set_pool_as_stale_with_wrong_type() {
    ExtBuilder::default().build().execute_with(|| {
        let idx = if let Asset::CategoricalOutcome(_, idx) = ASSET_A { idx
} else { 0 };
        create_initial_pool_with_funds_for_alice(ScoringRule::CPMM,
true);
        assert_ok!(Swaps::pool_join(alice_signed(), 0, _1, vec!(_1, _1,
_1, _1),));
        assert_ok!(Swaps::admin_set_pool_as_stale(
            Origin::root(),
            MarketType::Scalar(1..=2),
            0,
            OutcomeReport::Categorical(idx)
        ),);
    });
}
```

And the error would be:

```
thread 'tests::call_admin_set_pool_as_stale_with_wrong_type' panicked at
'Expected Ok(_). Got Err(
    Module {
        index: 5,
        error: 28,
        message: Some(
            "WinningAssetNotFound",
        ),
```

```
    },
)', zrml/swaps/src/tests.rs:52:9
```

Impact

Since only the user with the “Root Origin” role is able to use the "admin_set_pool_as_stale" and the "set_pool_as_stale" functions, this issue is marked as LOW.

Recommendation

It is recommended to maintain and retrieve the market type in the same way as the "pool_account_id" is retrieved.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.3 [prediction-markets] Minimum amount not required in "buy_complete_set"

Description	LOW

It was identified that no minimum amount is required in the publicly available dispatch functionality which is provided to buy a complete set of outcome shares of a market. Normally, when calling this function on a categorical market with five different outcomes, five different shares will be transferred to the callee. However, it was found that a user can issue requests for zero amounts.

The issue exists at the "buy_complete_set()" function:

File: zeitgeist/zrml/prediction-markets/src/lib.rs

```
278:         /// Buys the complete set of outcome shares of a market. For
example, when calling this
279:           /// function on a categorical market with five different
outcomes, five different shares
280:         /// will be transferred to the callee.
281:         ///
282:         /// The amount of each share will equal the provided `amount`
parameter.
283:         ///
284:         /// NOTE: This is the only way to create new shares.
285:          // Note: `buy_complete_set` weight consumption is dependent
on how many assets exists.
286:         // Unfortunately this information can only be retrieved with
a storage call, therefore
287:         // The worst-case scenario is assumed and the correct weight
is calculated at the end of this function.
288:         // This also occurs in numerous other functions.
289:         #[pallet::weight(
290:
T::WeightInfo::buy_complete_set(T::MaxCategories::get().into())
291:         )]
292:         #[transactional]
293:         pub fn buy_complete_set(
294:             origin: OriginFor<T>,
295:             market_id: MarketIdOf<T>,
```

```
296:             #[pallet::compact] amount: BalanceOf<T>,
297:         ) -> DispatchResultWithPostInfo {
298:             let sender = ensure_signed(origin)?;
299:             Self::do_buy_complete_set(sender, market_id, amount)
300:         }
```

And then the "do_buy_complete_set()" is called:

File: zeitgeist/zrml/prediction-markets/src/lib.rs

```
1396:         pub(crate) fn do_buy_complete_set(
1397:             who: T::AccountId,
1398:             market_id: MarketIdOf<T>,
1399:             amount: BalanceOf<T>,
1400:         ) -> DispatchResultWithPostInfo {
1401:              ensure!(CurrencyOf::<T>::free_balance(&who) >= amount,
Error::<T>::NotEnoughBalance);
1402:
1403:             let market = T::MarketCommons::market(&market_id)?;
1404:                  ensure!(market.scoring_rule == ScoringRule::CPMM,
Error::<T>::InvalidScoringRule);
1405:             Self::ensure_market_is_active(&market.period)?;
1406:             // The check below is primarily to ensure that the market
is
1407:             // not a pending advised market.
1408:                     ensure!(market.status == MarketStatus::Active,
Error::<T>::MarketIsNotActive);
1409:
1410:             let market_account = Self::market_account(market_id);
1411:             CurrencyOf::<T>::transfer(
1412:                 &who,
1413:                 &market_account,
1414:                 amount,
1415:                 ExistenceRequirement::KeepAlive,
1416:             )?;
1417:
1418:             let assets = Self::outcome_assets(market_id, &market);
1419:             for asset in assets.iter() {
1420:                 T::Shares::deposit(*asset, &who, amount)?;
1421:             }
1422:
```

```
1423:              Self::deposit_event(Event::BoughtCompleteSet(market_id,
amount, who));
1424:
1425:              let assets_len: u32 = assets.len().saturated_into();
1426:              let max_cats: u32 = T::MaxCategories::get().into();
1427:
Self::calculate_actual_weight(&T::WeightInfo::buy_complete_set,
assets_len, max_cats)
1428:          }
```

For example, in the following case, a zero amount is set and the test succeeds:

```
#[test]
fn buy_zero_shares() {
    ExtBuilder::default().build().execute_with(|| {
        assert_ok!(PredictionMarkets::create_categorical_market(
            Origin::signed(ALICE),
            BOB,
            MarketPeriod::Block(0..100),
            gen_metadata(2),
            MarketCreation::Advised,
            3,
            MarketDisputeMechanism::SimpleDisputes,
            ScoringRule::CPMM
        ));

assert_ok!(PredictionMarkets::approve_market(Origin::signed(SUDO), 0));
        deploy_swap_pool(MarketCommons::market(&0).unwrap(), 0).unwrap();

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(ALICE),   0,
0 * BASE));
        run_to_block(50);
        let market = MarketCommons::market(&0).unwrap();
        let assets = PredictionMarkets::outcome_assets(0, &market);
        for asset in assets.iter() {
            let bal = Tokens::free_balance(*asset, &ALICE);
            assert_eq!(bal, 0 * BASE);
        }
    });
```

```
}
```

Impact

If the weight is correctly calculated, the impact of this issue is limited. As a result, the risk of this issue is low.

Recommendation

It is advisable to ensure that the amount is bigger than zero.

```
ensure!(amount > 0, Error::<T>:: NotEnoughAssets);
```

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.4 [swaps] Admin functions without "deposit_event" at "swaps

Description	LOW

It was found that certain admin functions do not utilize the "deposit_event". The corresponding events are also missing. A pallet can emit events when it wants to notify external entities about changes or conditions in the runtime to external entities like users, chain explorers, or dApps.

More precisely, the admin function " admin_set_pool_as_stale" does not send the event. The issue exists at the following location:

```
File: /zrml/swaps/src/lib.rs
085:     #[pallet::call]
086:     impl<T: Config> Pallet<T> {
087:         #[pallet::weight(T::WeightInfo::admin_set_pool_as_stale())]
088:         #[frame_support::transactional]
089:         pub fn admin_set_pool_as_stale(
090:             origin: OriginFor<T>,
091:             market_type: MarketType,
092:             pool_id: PoolId,
093:             outcome_report: OutcomeReport,
094:         ) -> DispatchResult {
095:             ensure_root(origin)?;
096:             Self::set_pool_as_stale(
097:                 &market_type,
098:                 pool_id,
099:                 &outcome_report,
100:                 &Self::pool_account_id(pool_id),
101:             )?;
102:             Ok(())
103:         }
104:
```

And the “set_pool_as_stale” will not emit an event either:

```
File:
/Users/fisherman/Desktop/assessment1/zeitgeist/zrml/swaps/src/lib.rs
1605:
```

```
1606:         /// Pool will be marked as `PoolStatus::Stale`. If market is
categorical, removes everything
1607:         /// that is not ZTG or winning assets from the selected pool.
Additionally, it distributes
1608:         /// the rewards to all pool share holders.
1609:         ///
1610:         /// Does nothing if pool is already stale. Returns `Err` if
`pool_id` does not exist.
1611:         ///
1612:         /// # Arguments
1613:         ///
1614:         /// * `market_type`: Type of the market (e.g. categorical or
scalar).
1615:         /// * `pool_id`: Unique pool identifier associated with the
pool to be made stale.
1616:         /// * `outcome_report`: The resulting outcome.
1617:           /// * `winner_payout_account`: The account that exchanges
winning assets against rewards.
1618:         #[frame_support::transactional]
1619:         fn set_pool_as_stale(
1620:             market_type: &MarketType,
1621:             pool_id: PoolId,
1622:             outcome_report: &OutcomeReport,
1623:             winner_payout_account: &T::AccountId,
1624:         ) -> Result<Weight, DispatchError> {
1625:             let mut extra_weight = 0;
1626:             let mut total_assets = 0;
1627:
1628:             Self::mutate_pool(pool_id, |pool| {
1629:                 if pool.pool_status == PoolStatus::Stale {
1630:                     return Ok(());
1631:                 }
1632:
1633:                     ensure!(pool.pool_status == PoolStatus::Active,
Error::<T>::InvalidStateTransition);
1634:                 let base_asset = pool.base_asset;
1635:                 let mut winning_asset: Result<_, DispatchError> =
1636:                     Err(Error::<T>::WinningAssetNotFound.into());
1637:
1638:                 if let MarketType::Categorical(_) = market_type {
```

```
1639:                                         let  base_asset_or_default  =
base_asset.unwrap_or(Asset::Ztg);
1640:
1641:                 if let OutcomeReport::Categorical(winning_asset_idx)
= outcome_report {
1642:                         pool.assets.retain(|el| {
1643:                             if let Asset::CategoricalOutcome(_, idx)
= *el {
1644:                                 if idx == *winning_asset_idx {
1645:                                     winning_asset = Ok(*el);
1646:                                     return true;
1647:                                 };
1648:                             }
1649:
1650:                             *el == base_asset_or_default
1651:                         });
1652:                     }
1653:
1654:                     total_assets = pool.assets.len();
1655:                 }
1656:
1657:                 let winning_asset_unwrapped = winning_asset?;
1658:
1659:                                             if  pool.scoring_rule  ==
ScoringRule::RikiddoSigmoidFeeMarketEma {
1660:                   T::RikiddoSigmoidFeeMarketEma::destroy(pool_id)?;
1661:                                              let  distribute_weight  =
Self::distribute_pool_share_rewards(
1662:                         pool,
1663:                         pool_id,
1664:                   base_asset.ok_or(Error::<T>::BaseAssetNotFound)?,
1665:                         winning_asset_unwrapped,
1666:                         winner_payout_account,
1667:                     );
1668:                                                     extra_weight  =
extra_weight.saturating_add(T::DbWeight::get().writes(1));
1669:                                                     extra_weight  =
extra_weight.saturating_add(distribute_weight);
1670:                 }
1671:
1672:                 pool.pool_status = PoolStatus::Stale;
```

```
1673:                     Ok(())
1674:                 })?;
1675:
1676:
Ok(T::WeightInfo::set_pool_as_stale_without_reward_distribution(
1677:                     total_assets.saturated_into(),
1678:                 )
1679:                 .saturating_add(extra_weight))
1680:             }
1681:
```

Impact

Events are necessary to notify the off-chain world of successful state transitions. Administration functionalities should emit the corresponding events throughout the system's life cycle in order to provide credibility and confidence in the system.

Recommendation

It is recommended to emit an event related to this administrative functionality. For example:

```
Self::deposit_event(Event::PoolStale(poolid));
```

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4 Informational Findings

5.4.1 [prediction-markets] Unlisted dispatch function "deploy_swap_pool_and_additional_liquidity"

Description	INFO

It was identified that one of the available functions is not listed in the available Public Dispatches. Substrate uses Rust macros to aggregate the logic derived from pallets that are implemented for a runtime. In Substrate, the *"#[pallet::call]"* macro allows the developers to create dispatchable functions which will generate associated items from the "impl" code blocks. However, in the specific case it was found that one of the available functions is not included in the documentation of the examined pallet.

More precisely, according to the pallet documentation, the available Public Dispatches are the following:

File: zeitgeist/zrml/prediction-markets/src/lib.rs

```
23: //! #### Public Dispatches
24: //!
25: //! - `buy_complete_set` - Buys a complete set of outcome assets for a
market.
26: //! - `cancel_pending_market` - Allows the proposer of a market that
is currently in a `Proposed` state to cancel the market proposal.
27: //! - `create_categorical_market` - Creates a new categorical market.
28: //! - `create_cpmm_market_and_deploy_assets` - Create a market using
CPMM scoring rule, buy a complete set of the assets used and deploy.
29: //!     within and deploy an arbitrary amount of those that's greater
than the minimum amount.
30: //! - `create_scalar_market` - Creates a new scalar market.
31: //! - `deploy_swap_pool_for_market` - Deploys a single "canonical" pool
for a market.
32: //! - `dispute` - Submits a disputed outcome for a market.
33: //! - `global_dispute` - `unimplemented!()`
34: //! - `redeem_shares` - Redeems the winning shares for a market.
35: //! - `report` - Reports an outcome for a market.
```

```
36: //! - `sell_complete_set` - Sells a complete set of outcome assets for
a market.
```

However, the following "*deploy_swap_pool_and_additional_liquidity*" function is also available in the code:

File: zeitgeist/zrml/prediction-markets/src/lib.rs

```
626:           /// This function combines the creation of a market, the
buying of a complete set of
627:          /// outcome assets, the deployment of the minimum amount of
outcome assets and
628:         /// the optional deployment of additional outcome asset.
629:         ///
630:         /// # Arguments
631:         ///
632:         /// * `market_id`: Id of the market for that the pool should
be created and populated.
633:         /// * `amount_base_asset`: The amount of the base asset that
should be deployed.
634:         /// * `amount_outcome_assets`: A vector containing the amount
of each outcome asset that should be
635:          ///      deployed. The highest value will be used to buy a
complete set, i.e. every outcome
636:          ///     asset will be bought in quantities specified by the
highest value in this vector.
637:         ///     Any value that is lower than the highest value in the
vector signals that not
638:             ///       all assets should be deployed. For example,
`amount_outcome_assets = [120, 150]
639:         ///     means, that after deployment 30 of the first outcome
asset will be kept.
640:          /// * `weights`: The relative denormalized weight of each
asset price.
641:         #[pallet::weight(
642:
T::WeightInfo::buy_complete_set(T::MaxCategories::get().min(amount_outcom
e_assets.len().saturated_into()).into())
```

```
643:
.saturating_add(T::WeightInfo::deploy_swap_pool_for_market(T::MaxCategori
es::get().min(weights.len().saturated_into()).into()))
644:             // Overly generous estimation, since we have no access to
Swaps WeightInfo
645:              // (it is loosely coupled to this pallet using a trait).
Contains weight for
646:             // create_pool() and swap_exact_amount_in()
647:
.saturating_add(5_000_000_000.saturating_mul(T::MaxCategories::get().min(
amount_outcome_assets.len().saturated_into()).into()))
648:             .saturating_add(T::DbWeight::get().reads(2 as Weight))
649:         )]
650:         #[transactional]
651:         pub fn deploy_swap_pool_and_additional_liquidity(
652:             origin: OriginFor<T>,
653:             market_id: MarketIdOf<T>,
654:             amount_base_asset: BalanceOf<T>,
655:             amount_outcome_assets: Vec<BalanceOf<T>>,
656:             weights: Vec<u128>,
657:         ) -> DispatchResultWithPostInfo {
658:             let who = ensure_signed(origin.clone())?;
659:               // Buy a complete set of assets based on the highest
number to be deployed
660:                                                     let    assets    =
T::MarketCommons::market(&market_id)?.market_type;
661:             let zero_balance = <BalanceOf<T>>::zero();
662:             let max_assets = amount_outcome_assets
663:                 .iter()
664:                 .fold(zero_balance, |prev, cur| if prev > *cur { prev
} else { *cur });
665:              let weight_bcs = Self::buy_complete_set(origin.clone(),
market_id, max_assets)?
666:                 .actual_weight
667:                                                     .unwrap_or_else(||
T::WeightInfo::buy_complete_set(T::MaxCategories::get().into()));
668:             let weight_len = weights.len().saturated_into();
669:
670:             // Deploy a swap pool with MinLiquidity
671:          let _ = Self::deploy_swap_pool_for_market(origin, market_id,
weights)?;
```

```
672:             let pool_id = T::MarketCommons::market_pool(&market_id)?;
673:             let mut weight_pool_joins_and_sells = 0;
674:             let mut add_liqudity =
675:                 |amount: BalanceOf<T>, asset: Asset<MarketIdOf<T>>| -
> DispatchResult {
676:                                                 let local_weight =
T::Swaps::pool_join_with_exact_asset_amount(
677:                         who.clone(),
678:                         pool_id,
679:                         asset,
680:                         amount,
681:                         zero_balance,
682:                     )?;
683:                     weight_pool_joins_and_sells =
684:
weight_pool_joins_and_sells.saturating_add(local_weight);
685:                     Ok(())
686:                 };
687:
688:                     // Add additional liquidity as specified in
amount_outcome_assets
689:                                     for (idx, asset_amount) in
amount_outcome_assets.iter().enumerate() {
690:                 if *asset_amount == zero_balance {
691:                     continue;
692:                 };
693:
694:                 let remaining_amount =
695:
(*asset_amount).saturating_sub(MinLiquidity::get().saturated_into());
696:                 let asset_in = match assets {
697:                     MarketType::Categorical(_) => {
698:                                 Asset::CategoricalOutcome(market_id,
idx.saturated_into())
699:                     }
700:                     MarketType::Scalar(_) => {
701:                         if idx == 0 {
702:                                     Asset::ScalarOutcome(market_id,
ScalarPosition::Long)
703:                         } else {
```

```
704:                                     Asset::ScalarOutcome(market_id,
ScalarPosition::Short)
705:                             }
706:                         }
707:                 };
708:
709:                 if remaining_amount > zero_balance {
710:                     add_liqudity(remaining_amount, asset_in)?;
711:                 }
712:             }
713:
714:             // Add additional liquidity for the base asset
715:             let remaining_amount =
716:
(amount_base_asset).saturating_sub(MinLiquidity::get().saturated_into());
717:
718:             if remaining_amount > zero_balance {
719:                 add_liqudity(remaining_amount, Asset::Ztg)?;
720:             }
721:
722:             Ok(Some(
723:                 weight_bcs
724:
.saturating_add(T::WeightInfo::deploy_swap_pool_for_market(weight_len))
725:                     .saturating_add(weight_pool_joins_and_sells)
726:                     .saturating_add(T::DbWeight::get().reads(2)),
727:             )
728:             .into())
729:         }
```

Impact

This issue can make it more difficult to understand and maintain the product. It can make it more difficult and time-consuming to detect and/or fix vulnerabilities.

Recommendation

It is advisable to update the documentation and add the specific functionality.

CVSS Score
AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.2 [prediction-markets] Market Period Upper Limits not checked

Description	INFO

The team identified that no upper limits are defined for the "*range.end*" value of the Market Period. A user can create a Market with the higher possible value which is "u64:MAX". However, due to other calculations that take place at the "Report" functionality, reporting can be impossible due to overflow issues.

The issue exists at the following location:

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
931:         pub fn report(
932:             origin: OriginFor<T>,
933:             market_id: MarketIdOf<T>,
934:             outcome: OutcomeReport,
935:         ) -> DispatchResult {
...
948:                 let mut should_check_origin = false;
949:                 match market.period {
...
955:                     MarketPeriod::Timestamp(ref range) => {
956:                                         let rp_moment: MomentOf<T> =
T::ReportingPeriod::get().into();
957:                                 let reporting_period_in_ms = rp_moment *
MILLISECS_PER_BLOCK.into();
958:                                if T::MarketCommons::now() <= range.end +
reporting_period_in_ms {
```

```
959:                                should_check_origin = true;
960:                            }
961:                        }
962:                    }
...
989:          }
```

The MILLISECS_PER_BLOCK is 12000

```
File: zeitgeist/primitives/src/constants.rs
18: pub const MILLISECS_PER_BLOCK: u32 = 12000;
```

and the "T::ReportingPeriod::get()" is 7200

```
File:  zeitgeist/primitives/src/constants.rs
79:     pub const ReportingPeriod: u32 = BLOCKS_PER_DAY as _;
```

As a result, the multiplication would be 86400000. The u64::Max is 18446744073709551615. As a result, any range bigger than u64::MAX-86400000 will lead to overflow or to stop the origin validation when the market is closed.

For example, if a timestamp range is used, this case will work, while any other Timestamp range with higher "*range.end*" will fail:

```
#[test]
fn test_max_endrange_timestamp() {
    ExtBuilder::default().build().execute_with(|| {
        // Creates a permissionless market.
        assert_ok!(PredictionMarkets::create_categorical_market(
            Origin::signed(ALICE),
            BOB,
            MarketPeriod::Timestamp(0..u64::MAX-86400000),
            gen_metadata(2),
```

```
            MarketCreation::Permissionless,
            2,
            MarketDisputeMechanism::SimpleDisputes,
            ScoringRule::CPMM
        ));

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(BOB),     0,
CENT));
        // set the timestamp
        Timestamp::set_timestamp(u64::MAX);
        assert_ok!(PredictionMarkets::report(
            Origin::signed(BOB),
            0,
            OutcomeReport::Categorical(1)
        ));
    });
}
```

The same issue will occur for any higher block "*range.end*" value than u64:MAX-7200.

```
#[test]
fn test_max_endrange_blocks() {
    ExtBuilder::default().build().execute_with(|| {
        // Creates a permissionless market.
        assert_ok!(PredictionMarkets::create_categorical_market(
            Origin::signed(ALICE),
            BOB,
            MarketPeriod::Block(0..u64::MAX-7200),
            gen_metadata(2),
            MarketCreation::Permissionless,
            2,
            MarketDisputeMechanism::SimpleDisputes,
            ScoringRule::CPMM
        ));

assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(BOB),     0,
CENT));
        // set the timestamp
```

```
        run_to_block(u64::MAX-7200);
        assert_ok!(PredictionMarkets::report(
            Origin::signed(BOB),
            0,
            OutcomeReport::Categorical(1)
        ));
    });
}
```

Impact

The issue is marked as INFORMATIONAL as both cases are not expected to occur in the created markets.

Recommendation

It is advisable to set upper limits on market creation for the Blocks or the Timestamp "*range.end*" parameter, that will take into account the "ReportingPeriod".

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.3 [prediction-markets] Incorrect start range in "admin_move_market_to_closed"

Description	INFO

It was found that the "admin_move_market_to_closed" does not handle cases where the "current_block" or the " T::MarketCommons::now()" is smaller than the "range.start" value. As a result, it is possible that when the admin uses the specific functionality, the market that will be mutated will have an invalid range.

The issue exists at the following location:

File: zeitgeist/zrml/prediction-markets/src/lib.rs

```
193:
#[pallet::weight(T::WeightInfo::admin_move_market_to_closed())]
194:         pub fn admin_move_market_to_closed(
195:             origin: OriginFor<T>,
196:             market_id: MarketIdOf<T>,
197:         ) -> DispatchResult {
198:             T::CloseOrigin::ensure_origin(origin)?;
199:             T::MarketCommons::mutate_market(&market_id, |m| {
200:                 m.period = match m.period {
201:                     MarketPeriod::Block(ref range) => {
202:                                                 let    current_block    =
<frame_system::Pallet<T>>::block_number();
203:                      MarketPeriod::Block(range.start..current_block)
204:                     }
205:                     MarketPeriod::Timestamp(ref range) => {
206:                         let now = T::MarketCommons::now();
207:                         MarketPeriod::Timestamp(range.start..now)
208:                     }
209:                 };
210:                 Ok(())
211:             })?;
212:             Ok(())
213:         }
```

For example, the following test case can be used to replicate the issue

```
#[test]
fn the_entire_market_lifecycle_works_with_timestamps() {
    ExtBuilder::default().build().execute_with(|| {
        // Creates a permissionless market.
        assert_ok!(PredictionMarkets::create_categorical_market(
            Origin::signed(ALICE),
            BOB,
            MarketPeriod::Timestamp(u64::MAX-1..u64::MAX),
            gen_metadata(2),
            MarketCreation::Permissionless,
            2,
            MarketDisputeMechanism::SimpleDisputes,
            ScoringRule::CPMM
        ));
      assert_ok!(PredictionMarkets::buy_complete_set(Origin::signed(BOB),
0, CENT));

assert_ok!(PredictionMarkets::admin_move_market_to_closed(Origin::signed(
SUDO), 0));
        assert_ok!(PredictionMarkets::report(
            Origin::signed(BOB),
            0,
            OutcomeReport::Categorical(1)
        ));
    });
}
```

The expected mutation for the Market period will be:

MarketPeriod::Timestamp(u64::MAX-1..0)

Impact

Since the "range.start" is currently not used in any security decision, the issue is marked as INFORMATIONAL

Recommendation

If the market range is used in a future security decision, it is advisable to always validate the market range for such cases or change the specific functionality to set a valid range.

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

6 Retest Results

6.1 Retest of High Severity Findings

6.1.1 [authorized] Lack of support for common authorized user in multiple markets at "authorize_market_outcome" call

Tested on 2022-05-18 by Chaintroopers and the issue was found to be fixed.

- github pull #574
- commit ids "141642a9ad6381ed5b37e2fc576d4892369ca279"):

File: zeitgeist/zrml/authorized/src/lib.rs

```
52:         pub fn authorize_market_outcome(
53:             origin: OriginFor<T>,
54:             market_id: MarketIdOf<T>,
55:             outcome: OutcomeReport,
56:         ) -> DispatchResult {
57:             let who = ensure_signed(origin)?;
58:             let market = T::MarketCommons::market(&market_id)?;
59:                         ensure!(market.status  ==  MarketStatus::Disputed,
Error::<T>::MarketIsNotDisputed);
60:                          ensure!(market.matches_outcome_report(&outcome),
Error::<T>::OutcomeMismatch);
61:             if let MarketDisputeMechanism::Authorized(ref account_id) =
market.mdm {
62:                 if account_id != &who {
63:                                                                     return
Err(Error::<T>::NotAuthorizedForThisMarket.into());
64:                 }
65:             } else {
66:                                                                     return
Err(Error::<T>::MarketDoesNotHaveDisputeMechanismAuthorized.into());
67:             }
68:             AuthorizedOutcomeReports::<T>::insert(market_id, outcome);
69:             Ok(())
70:         }
71:     }
72:
```

The following test case now works correctly:

```
#[test]
fn authorize_market_outcome_inserts_a_new_outcome_when_two_markets_forced()
{
    ExtBuilder::default().build().execute_with(|| {
        let market = market_mock::<Runtime>(ALICE);
        Markets::<Runtime>::insert(0, &market);
        Markets::<Runtime>::insert(1, market_mock::<Runtime>(BOB));
        Markets::<Runtime>::mutate(1, |el| {
            el.as_mut().unwrap().mdm                                    =
MarketDisputeMechanism::Authorized(ALICE);
        });

assert_ok!(Authorized::authorize_market_outcome(Origin::signed(ALICE),    0,
OutcomeReport::Scalar(3)));
        assert_eq!(
            Authorized::on_resolution(&[], &0, &market).unwrap(),
            Some(OutcomeReport::Scalar(3))
        );
    });
}
```

The output will be:

```
└─$ cargo test authorize_market_outcome_inserts_a_new_outcome_when_two_markets_forced
   Compiling zrml-authorized v0.3.2 (/home/kali/Desktop/retestzeitgeist/new/zeitgeist/zrml/authorized)
    Finished test [unoptimized + debuginfo] target(s) in 12.70s
     Running unittests src/lib.rs (/home/kali/Desktop/retestzeitgeist/new/zeitgeist/target/debug/deps/zrml_authorized-21755235e7eaee5c)

running 1 test
test tests::authorize_market_outcome_inserts_a_new_outcome_when_two_markets_forced ... ok
```

As a result. the status is marked as CLOSED.

6.1.2 [prediction-markets] A permissionless market can be rejected in "reject_market"

Tested on 2022-05-14 by Chaintroopers and the issue was found to be fixed

- github pull #585
- commit id "8200a3513c245680e2f5bffb0ae6d840db777ffa"

The "reject_market" cannot be used in a market that is not in proposed state. Since a permissionless market would not be in this state, the Approval Origin would also not be able to reject it:

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
934:         pub fn reject_market(
935:             origin: OriginFor<T>,
936:             #[pallet::compact] market_id: MarketIdOf<T>,
937:         ) -> DispatchResult {
938:             T::ApprovalOrigin::ensure_origin(origin)?;
939:             let market = T::MarketCommons::market(&market_id)?;
940:                     ensure!(market.status == MarketStatus::Proposed, Error::<T>::InvalidMarketStatus);
```

The test case now correctly fails:

```
----  tests::it_allows_the_advisory_origin_to_reject_permissionless_markets
stdout ----
thread
'tests::it_allows_the_advisory_origin_to_reject_permissionless_markets'
panicked at 'Expected Ok(_). Got Err(
    Module(
        ModuleError {
            index: 6,
            error: 26,
            message: Some(
                "InvalidMarketStatus",
            ),
        },
    ),
)', zrml/prediction-markets/src/tests.rs:100:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As a result. the status is marked as CLOSED.

6.1.3 [prediction-markets] Market state is ignored in "Report" function

Tested on 2022-05-14 by Chaintroopers and the issue was found to be fixed

- github pull #577
- commit ids "c62257d338719c687995ef5b43702797a8db2666"

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
1554:         fn ensure_market_is_closed(
1555:            market: &Market<T::AccountId, T::BlockNumber, MomentOf<T>>,
1556:         ) -> DispatchResult {
1557:                         ensure!(market.status == MarketStatus::Active,
Error::<T>::MarketIsNotClosed);
1558:             ensure!(
1559:                 match &market.period {
1560:                     MarketPeriod::Block(range) => {
1561:                          <frame_system::Pallet<T>>::block_number() >=
range.end
1562:                     }
1563:                     MarketPeriod::Timestamp(range) => {
1564:                         T::MarketCommons::now() >= range.end
1565:                     }
1566:                 },
1567:                 Error::<T>::MarketIsNotClosed
1568:             );
```

The test case now correctly fails:

```
---- tests::check_proposed_reported_canceled stdout ----
thread 'tests::check_proposed_reported_canceled' panicked at 'Expected
Ok(_). Got Err(
    Module(
        ModuleError {
            index: 6,
            error: 10,
            message: Some(
                ""MarketIsNotClosed"",
            ),
        },
    ),
)', zrml/prediction-markets/src/tests.rs:77:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As a result. the status is marked as CLOSED.

6.1.4 [prediction-markets] Missing transactional annotation in "create_categorical_market"

Tested on 2022-05-14 by Chaintroopers and the issue was found to be fixed

- github pull #576
- commit id "016c20803976c045de9bfce148ffb931f3f66fa5"

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
391:         #[transactional]
392:         pub fn create_categorical_market(
```

As a result, the status is marked as CLOSED.

6.2 Retest of Medium Severity Findings

6.2.1 [prediction-markets] Market state ignored in "reject_market" function

Tested on 2022-05-14 by Chaintroopers and the issue was found to be fixed

- github pull #585
- commit ids "8200a3513c245680e2f5bffb0ae6d840db777ffa"

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
934:         pub fn reject_market(
935:             origin: OriginFor<T>,
936:             #[pallet::compact] market_id: MarketIdOf<T>,
937:         ) -> DispatchResult {
938:             T::ApprovalOrigin::ensure_origin(origin)?;
939:             let market = T::MarketCommons::market(&market_id)?;
940:                     ensure!(market.status == MarketStatus::Proposed,
Error::<T>::InvalidMarketStatus);
```

The test case now correctly fails:

```
---- tests::reject_approved_market stdout ----
thread 'tests::reject_approved_market' panicked at 'Expected Ok(_). Got Err(
    Module(
        ModuleError {
            index: 6,
            error: 26,
            message: Some(
                "InvalidMarketStatus",
            ),
        },
    ),
)', zrml/prediction-markets/src/tests.rs:104:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As a result. the status is marked as CLOSED.

6.2.2 [prediction-markets] Reject market does not manage the related outcome assets

Tested on 2022-05-14 by Chaintroopers and the issue was found to be fixed

- github pull #585
- commit ids "8200a3513c245680e2f5bffb0ae6d840db777ffa"

The "reject_market" cannot be used in a market that is not in proposed state and may have outcome assets:

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
934:         pub fn reject_market(
935:             origin: OriginFor<T>,
936:             #[pallet::compact] market_id: MarketIdOf<T>,
937:         ) -> DispatchResult {
938:             T::ApprovalOrigin::ensure_origin(origin)?;
939:             let market = T::MarketCommons::market(&market_id)?;
940:                         ensure!(market.status == MarketStatus::Proposed,
Error::<T>::InvalidMarketStatus);
```

The test case now correctly fails:

```
---- tests::reject_doesnot_manage_outcome_assets stdout ----
thread 'tests::reject_doesnot_manage_outcome_assets' panicked at 'Expected
Ok(_). Got Err(
    Module(
        ModuleError {
            index: 6,
            error: 26,
            message: Some(
                "InvalidMarketStatus",
            ),
        },
    ),
)', zrml/prediction-markets/src/tests.rs:89:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As a result. the status is marked as CLOSED.

6.2.3 [swaps] Minimum amount not required in "pool_join_subsidy"

Tested on 2022-05-14 by Chaintroopers and the issue was found to be fixed

- github issue #601
- commit ids "f3c2ef17ed180da1815e88915fa6d6521deb4b4a"" and ""098a363b379d43622dbc900ee077b37019c7a90d"

```
File: zeitgeist/zrml/swaps/src/lib.rs
412:   ensure!(amount != Zero::zero(), Error::<T>::ZeroAmount);
413:
...
434:    ensure!(
435.      >= T::MinSubsidyPerAccount::get(),
436:                                  Error::<T>::InvalidSubsidyAmount
437:                              );
```

The provided test case now correctly fails:

```
---- tests::test_subsidy_provider stdout ----
thread 'tests::test_subsidy_provider' panicked at 'Expected Ok(_). Got Err(
    Module(
        ModuleError {
            index: 5,
            error: 30,
            message: Some(
                "ZeroAmount",
            ),
        },
    ),
)', zrml/swaps/src/tests.rs:70:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As a result, the status is marked as CLOSED.

6.3 Retest of Low Severity Findings

6.3.1 [prediction-markets] Admin functions without "deposit_event"

Tested on 2022-06-28 by Chaintroopers and the issue was found to be fixed

- github pull #651
- commit ids "a50eb594fafda5bd793afd41dd34673b14ec0b69", "bef0b3461dfcce4f81fc361f729eefa340bf19d1"

```
File:  /zeitgeist/zrml/prediction-markets/src/lib.rs
228:         #[pallet::weight(T::WeightInfo::admin_move_market_to_closed())]
229:          #[transactional]
230:          pub fn admin_move_market_to_closed(
231:              origin: OriginFor<T>,
232:              #[pallet::compact] market_id: MarketIdOf<T>,
233:          ) -> DispatchResult {
234:              // TODO(#638): Handle Rikiddo markets!
235:              T::CloseOrigin::ensure_origin(origin)?;
236:              let market = T::MarketCommons::market(&market_id)?;
237:              Self::ensure_market_is_active(&market)?;
238:              Self::clear_auto_close(&market_id)?;
239:              Self::close_market(&market_id)?;
240:              Ok(())
...
1587:              pub(crate) fn close_market(market_id: &MarketIdOf<T>) ->
Result<Weight, DispatchError> {
1588:              T::MarketCommons::mutate_market(market_id, |market| {
1589:                          ensure!(market.status == MarketStatus::Active,
Error::<T>::InvalidMarketStatus);
1590:                  market.status = MarketStatus::Closed;
1591:                  Ok(())
1592:              })?;
...
1598:              Self::deposit_event(Event::MarketClosed(*market_id));
....
1600:              Ok(total_weight)
1601:          }
1602:
```

Regarding the "admin_move_market_to_closed", the event has been added.

```
File: /zeitgeist/zrml/prediction-markets/src/lib.rs
253:         #[transactional]
254:         pub fn admin_move_market_to_resolved(
255:             origin: OriginFor<T>,
256:             #[pallet::compact] market_id: MarketIdOf<T>,
257:         ) -> DispatchResultWithPostInfo {
258:             T::ResolveOrigin::ensure_origin(origin)?;
260:             ...
265:             Self::clear_auto_resolve(&market_id)?;
266:             let market = T::MarketCommons::market(&market_id)?;
267:             let weight = Self::on_resolution(&market_id, &market)?;
268:             Ok(Some(weight).into())
269:         }
...
1615:          fn on_resolution(
1616:              market_id: &MarketIdOf<T>,
1617:             market: &Market<T::AccountId, T::BlockNumber, MomentOf<T>>,
1618:          ) -> Result<u64, DispatchError> {
...
1745:              Self::deposit_event(Event::MarketResolved(
1746:                  *market_id,
1747:                  MarketStatus::Resolved,
1748:                  resolved_outcome,
1749:              ));
....
1757:          }
1758:
```

In reference to the "admin_move_market_to_resolved", there is no event to indicate the admin operation, but the "MarketStatus::Resolved" even can be used to identify the action

As a result, the status is marked as CLOSED.

6.3.2 [swaps] User defined market type in "admin_set_pool_as_stale" at "swaps"

Tested on 2022-05-14 by Chaintroopers and the issue was found to be fixed

- github issue #582
- commit ids ""a9ee912feac633dbf2ff8bf55a15680d53f1ebde"

```
File: zeitgeist/zrml/swaps/src/lib.rs
90:         pub fn admin_set_pool_to_stale(
91:             origin: OriginFor<T>,
92:             #[pallet::compact] market_id: <<T as Config>::MarketCommons
as MarketCommonsPalletApi>::MarketId,
93:             outcome_report: OutcomeReport,
94:         ) -> DispatchResult {
```

The test case now correctly fails (function was renamed to "admin_set_pool_to_stale"):

```
This function takes 3 arguments but 4 arguments were supplied
  --> zrml/swaps/src/tests.rs:70:20
   |
70 |         assert_ok!(Swaps::admin_set_pool_to_stale(
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected 3 arguments
71 |             Origin::root(),
   |             --------------
72 |             MarketType::Scalar(1..=2),
   |             ------------------------
73 |             0,
   |             -
74 |             OutcomeReport::Categorical(idx)
   |             ------------------------------ supplied 4 arguments
   |
```

As a result. the status is marked as CLOSED.

6.3.3 [prediction-markets] Minimum amount not required in "buy_complete_set"

Tested on 2022-05-14 by Chaintroopers and the issue was found to be fixed

- github issue #561
- commit id "62a677731ee527fade918b92af88b116024a2b6e"

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
1427:                             ensure!(amount  !=  BalanceOf::<T>::zero(),
Error::<T>::ZeroAmount);
1428:                 ensure!(CurrencyOf::<T>::free_balance(&who)  >=  amount,
Error::<T>::NotEnoughBalance);
```

The test case now correctly fails:

```
---- tests::buy_zero_shares stdout ----
thread 'tests::buy_zero_shares' panicked at 'Expected Ok(_). Got Err(
    DispatchErrorWithPostInfo {
        post_info: PostDispatchInfo {
            actual_weight: None,
            pays_fee: Pays::Yes,
        },
        error: Module(
            ModuleError {
                index: 6,
                error: 27,
                message: Some(
                    "ZeroAmount",
                ),
            },
        ),
    },
)', zrml/prediction-markets/src/tests.rs:82:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As a result, the status is marked as CLOSED.

6.3.4 [swaps] Admin functions without "deposit_event" at "swaps

Tested on 2022-06-28 by Chaintroopers and the issue was found to be fixed

- github issue #651
- commit id "3f3bbad6a2b82040e28d5aaa33d91d42a8e8121d"

```
File: /zeitgeist/zrml/swaps/src/lib.rs
1799:         #[frame_support::transactional]
1800:         fn clean_up_pool(
1801:             market_type: &MarketType,
1802:             pool_id: PoolId,
1803:             outcome_report: &OutcomeReport,
1804:             winner_payout_account: &T::AccountId,
1805:         ) -> Result<Weight, DispatchError> {
1806:             let mut weight = 0;
1807:             Self::mutate_pool(pool_id, |pool| {
1808:                     ensure!(pool.pool_status == PoolStatus::Closed,
Error::<T>::InvalidStateTransition);
1809:                 pool.pool_status = PoolStatus::Clean;
1810:                 Ok(())
1811:             })?;
1812:             if let MarketType::Categorical(_) = market_type {
1813:                                                             weight    =
weight.saturating_add(Self::clean_up_pool_categorical(
1814:                     pool_id,
1815:                     outcome_report,
1816:                     winner_payout_account,
1817:                 )?);
1818:             }
1819:             Self::deposit_event(Event::<T>::PoolCleanedUp(pool_id));
1820:             // (No extra work required for scalar markets!)
1821:             Ok(weight)
1822:         }
1823:
1824:         /// Swap - Exact amount in
```

Function "set_pool_to_stale" is now replaced by "admin_clean_up_pool" and "clean_up_pool". As a result, the status is marked as CLOSED.

6.4 Retest of Informational Findings

6.4.1 [prediction-markets] Unlisted dispatch function "deploy_swap_pool_and_additional_liquidity"

Tested on 2022-05-14 by Chaintroopers and the issue was found to be fixed

- github issue #619
- commit id "f160df076b1a07a72c7e1a0df94560e6b3c13243"

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
33: //! - `deploy_swap_pool_and_additional_liquidity` - Deploys a single ""canonical"" pool for a market,
34: //!   buys a complete set of the assets used and deploys the funds as specified.
```

As a result, the status is marked as CLOSED.

6.4.2 [prediction-markets] Market Period Upper Limits not checked

Tested on 2022-05-18 by Chaintroopers and the issue was found to be fixed

- github pull #614
- commit ids "481ec2ee2da3e30ac11b6319d092abf179c23c24"

```
File: zeitgeist/zrml/prediction-markets/src/lib.rs
1564:         fn ensure_market_period_is_valid(
1565:             period: &MarketPeriod<T::BlockNumber, MomentOf<T>>,
1566:         ) -> DispatchResult {
1567:             let verify = |start: u64, end: u64| -> DispatchResult {
1568:                 ensure!(start < end, Error::<T>::InvalidMarketPeriod);
1569:                             ensure!(end  <=  T::MaxMarketPeriod::get(),
Error::<T>::InvalidMarketPeriod);
1570:                 Ok(())
1571:             };
1572:             match period {
1573:                 MarketPeriod::Block(ref range) => {
1574:                                     verify(range.start.saturated_into(),
range.end.saturated_into())
1575:                 }
1576:                 MarketPeriod::Timestamp(ref range) => {
1577:                                     verify(range.start.saturated_into(),
range.end.saturated_into())
1578:                 }
1579:             }?;
1580:             Ok(())
1581:         }
```

The test cases now correctly fail:

```
---- tests::test_max_endrange_timestamp stdout ----
thread 'tests::test_max_endrange_timestamp' panicked at 'Expected Ok(_). Got
Err(
    DispatchErrorWithPostInfo {
        post_info: PostDispatchInfo {
            actual_weight: None,
            pays_fee: Pays::Yes,
        },
        error: Module(
            ModuleError {
                index: 6,
                error: 27,
                message: Some(
                    "InvalidMarketPeriod",
                ),
            },
        ),
    },
)', zrml/prediction-markets/src/tests.rs:38:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
---- tests::test_max_endrange_blocks stdout ----
thread 'tests::test_max_endrange_blocks' panicked at 'Expected Ok(_). Got
Err(
    DispatchErrorWithPostInfo {
        post_info: PostDispatchInfo {
            actual_weight: None,
            pays_fee: Pays::Yes,
        },
        error: Module(
            ModuleError {
                index: 6,
                error: 27,
                message: Some(
                    "InvalidMarketPeriod",
                ),
            },
        ),
    },
)', zrml/prediction-markets/src/tests.rs:64:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As a result. the status is marked as CLOSED.

6.4.3 [prediction-markets] Incorrect start range in "admin_move_market_to_closed"

Tested on 2022-06-28 by Chaintroopers and the issue was found to be fixed

- github pull #651
- commit id "67bd8ff71b8e70f9b1f091159ca355cc2793b125"

The function now validates if the market is active:

```
File: /zeitgeist/zrml/prediction-markets/src/lib.rs
228:         #[pallet::weight(T::WeightInfo::admin_move_market_to_closed())]
229:          #[transactional]
230:          pub fn admin_move_market_to_closed(
231:              origin: OriginFor<T>,
232:              #[pallet::compact] market_id: MarketIdOf<T>,
233:          ) -> DispatchResult {
234:              // TODO(#638): Handle Rikiddo markets!
235:              T::CloseOrigin::ensure_origin(origin)?;
236:              let market = T::MarketCommons::market(&market_id)?;
237:              Self::ensure_market_is_active(&market)?;
238:              Self::clear_auto_close(&market_id)?;
239:              Self::close_market(&market_id)?;
240:              Ok(())
241:          }
242:
```

As a result, the status is marked as CLOSED.

References & Applicable Documents

Ref.	Title	Version
N/A	*N/A*	*N/A*

Document History

Revision	Description	Changes Made By	Date
0.2	*Initial Draft*	*Chaintroopers*	*May 2nd, 2022*
1.0	*Final Report*	*Chaintroopers*	*May 3rd, 2022*
1.1	*Retest Report*	*Chaintroopers*	*May 19th, 2022*
1.2	*Retest Report*	*Chaintroopers*	*June 28th, 2022*