AUDIT REPORT
9Lives Security Review

Conducted by DadeKuma

16th December - 22nd December 2024

Table of contents

Introduction

A time-boxed security review of 9Lives was done by **DadeKuma**, with a focus on the security aspects of the application's smart contracts implementation.

About DadeKuma

DadeKuma is an independent smart contract security researcher specialized in EVM and Rust-based protocols and blockchains. With a proven track record that includes multiple first-place finishes on public competition platforms like Code4rena, **DadeKuma** has consistently demonstrated expertise in identifying critical vulnerabilities across various blockchain projects. Dedicated to enhancing the security of the blockchain ecosystem, he actively engages in in-depth audits and research.

Explore his work on GitHub, or connect via X or Telegram.

About 9Lives

9Lives is the most capital-efficient prediction market developed by the Superposition team. Inspired by decentralized finance, 9Lives combines parimutuel markets with a continuous double auction, providing deep liquidity and a risk-free environment for market participants.

Security Review Disclaimer

While a smart contract security review aims to identify vulnerabilities, it cannot eliminate all risks. This process is limited by the time, resources, and expertise, and absolute security cannot be ensured. No responsibility is taken for any issues, losses, or damages that may arise after the review, whether vulnerabilities are identified or not. To strengthen the security of a project, it is strongly recommended to conduct additional audits, launch bug bounty programs, and implement ongoing on-chain monitoring.

Risk Classification

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

Impact

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

Likelihood

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

Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed).
- High - Must fix (before deployment if not already deployed).
- Medium - Should fix.
- Low - Could fix.

Scope

Review commit hash: b0a9ca24636f244705e55800ad141ab9392297c6

The following files were in scope of the audit:

- `src/contract_infra_market.rs`
- `src/contract_lockup.rs`
- `src/lockup_call.rs`
- `src/storage_infra_market.rs`
- `src/storage_lockup.rs`
- `src/timing_infra_market.rs`
- `src/contract_trading_mint.rs`

Findings

ID	Title	Severity	Status
C-01	Winners will be slashed instead of losers	Critical	Fixed
C-02	Wrongly whinged campaigns cannot be settled	Critical	Fixed
C-03	Lockup contract can be reinitialized by anyone	Critical	Fixed
C-04	DPM trading uses a wrong amount to calculate total shares	Critical	Fixed
C-05	Sweep users can't be paid as funds are permanently frozen	Critical	Fixed
C-06	Existing commits will be overridden by other users	Critical	Fixed
C-07	Traders can extend the trading period to continue until after the winner is decided	Critical	Fixed
C-08	Traders can mint shares even after the result is decided	Critical	Fixed
H-01	Campaigns cannot be escaped after the call deadline	High	Fixed
H-02	Fees for the team are never collected	High	Fixed
M-01	MEV bots are incentivized to randomly call campaigns once they are registered	Medium	Fixed
M-02	Epochs can be increased before the end of anything goes period	Medium	Fixed
L-01	Contracts can't be disabled	Low	Fixed
L-02	`are_we_after_whinging_period` uses a wrong parameter	Low	Acknowledged
L-03	Inframarket periods are overlapping	Low	Acknowledged

[C-01] Winners will be slashed instead of losers

Severity

Impact: High, Loss of funds for winners instead of losers, as they get slashed by sweepers.

Likelihood: High, It will happen without preconditions.

Description

In `contract_infra_market::sweep`, users can slash losers who bet on the wrong outcome. However, the logic is inverted, causing the opposite to happen: only winners can be slashed, while losers cannot.

Recommendations

Consider the following change:

```
    // Let's check that they bet correctly. If they did, then we should
    // revert, since the caller made a mistake.
    assert_or!(
        !e.commitments.get(victim_addr).is_zero()
-           && e.reveals.get(victim_addr) == e.campaign_winner.get(),
+           && e.reveals.get(victim_addr) != e.campaign_winner.get(),
        Error::BadVictim
    );
```

Mitigation

The issue was fixed as recommended.

[C-02] Wrongly whinged campaigns cannot be settled

Severity

Impact: High, loss of funds for traders (funds are permanently locked), as the campaign winner can't be decided.

Likelihood: High, it will happen when the whinger calls a wrong outcome.

Description

In `contract_infra_market::sweep`, there is a check that ensures that the TX will fail if the whinger is wrong, but this should be a valid scenario (in this case, they would simply lose their bond).

This has several side effects, as a campaign cannot be swept, so it will remain in an inconclusive state. Moreover, losers cannot be slashed, resulting in a loss of funds for traders, as they aren't able to burn their shares.

Recommendations

Consider the following change:

```
    e.campaign_winner.set(voting_power_winner);
    e.campaign_winner_set.set(true);
-   assert_eq!(
-       voting_power_winner,
-       e.campaign_whinger_preferred_winner.get()
-   );
    // If the whinger was correct, we owe them their bond.
    if voting_power_winner == e.campaign_whinger_preferred_winner.get() {
        fusdc_call::transfer(e.campaign_who_whinged.get(),
BOND_FOR_WHINGE)?;
    }
```

Mitigation

The issue was fixed as recommended.

[C-03] Lockup contract can be reinitialized by anyone

Severity

Impact: High, anyone can put in a fake contract able to steal user funds when they try to make a deposit.

Likelihood: High, it will happen without preconditions.

Description

In `contract_lockup::ctor`, there is a permissionless contract initialization.

The issue is that this function can be called multiple times instead of just once. After some time when the contract is live (with correct values), any user can re-initialize the contract to let users interact with a fake infra market and/or token instead of the legit ones.

This would lead to loss of funds as users might deposit funds on a rugging contract.

Recommendations

Consider the following change to make sure that the contract can't be re-initialized:

```
     pub fn ctor(&mut self, token_impl: Address, infra_market: Address) ->
Result<(), Error> {
        assert_or!(!self.created.get(), Error::AlreadyConstructed);
        let token =
            proxy::deploy_proxy(token_impl).map_err(|_|
Error::NinelivesLockedArbCreateError)?;
        nineliveslockedarb_call::ctor(token, contract::address())?;
        evm::log(events::LockupTokenProxyDeployed { token });
        self.token_addr.set(token);
        self.infra_market_addr.set(infra_market);
        self.enabled.set(true);
```

```
+       self.created.set(true);
        Ok(())
    }
```

Mitigation

The issue was fixed as recommended.

[C-04] DPM trading uses a wrong amount to calculate total shares

Severity

Impact: High, loss of funds as more shares than intended can be minted.

Likelihood: High, it will happen without preconditions.

Description

In `contract_trading_mint::internal_mint`, the global amounts invested are calculated before the DPM shares are:

```
    self.outcome_invested.setter(outcome_id).set(
        outcome_invested_before
            .checked_add(value)
            .ok_or(Error::CheckedAddOverflow)?,
    );
    self.global_invested.set(
        self.global_invested
            .get()
            .checked_add(value)
            .ok_or(Error::CheckedAddOverflow)?,
    );
    // We need to increase by the amount we allocate, the AMM should do
that
    // internally. Some extra fields are needed for the DPM's
calculations.
    #[cfg(feature = "trading-backend-dpm")]
    let shares = {
        let shares_before = self.outcome_shares.get(outcome_id);
->      let shares = self.internal_dpm_mint(outcome_id, value)?; //@audit
uses the new invested values, but this should use the old ones
        self.outcome_shares.setter(outcome_id).set(
            shares_before
                .checked_add(shares)
                .ok_or(Error::CheckedAddOverflow)?,
        );
        self.global_shares.set(
            self.global_shares
```

```
                .get()
                .checked_add(shares)
                .ok_or(Error::CheckedAddOverflow)?,
        );
        shares
    };
```

The issue is that `internal_dpm_mint` uses these new global values, but the total share math should use the old ones. This can result in more shares minted than expected.

Recommendations

Consider calculating the `outcome_invested` and `global_invested` after the dpm shares are calculated, not before.

Mitigation

The issue was fixed as recommended.

[C-05] Sweep users can't be paid as funds are permanently frozen

Severity

Impact: High, loss of funds for the sweeper, funds are frozen permanently inside the contract.

Likelihood: High, it will happen when `on_behalf_of_addr == fee_recipient_addr` (user wants to receive on their own address).

Description

In `contract_infra_market::sweep` the victim will be `confiscated` if they committed a wrong outcome, and their staked position will be sent to the `fee_recipient_addr`:

```
    let confiscated: Uint<256, 4> =
->      lockup_call::confiscate(self.lockup_addr.get(), victim_addr,
contract::address())?;
    if confiscated.is_zero() {
        return Ok((caller_yield_taken, on_behalf_of_yield_taken));
    }
    // Now we send the confiscated amount and a small fee to the
    // caller for being first.
    if on_behalf_of_addr == fee_recipient_addr {
        on_behalf_of_yield_taken +=
->          lockup_call::confiscate(self.lockup_addr.get(), victim_addr,
on_behalf_of_addr)?;
```

The issue is that `confiscate` is called twice instead of using the cached result: the second call of `confiscate` will yield zero (as the user's staked amount is zero after the first call), so the sweeper will get nothing in return. Funds would be stuck in the contract due to the first `confiscate` call.

Recommendations

Consider the following change:

```
    let confiscated: Uint<256, 4> =
        lockup_call::confiscate(self.lockup_addr.get(), victim_addr,
contract::address())?;
    if confiscated.is_zero() {
        return Ok((caller_yield_taken, on_behalf_of_yield_taken));
    }
    // Now we send the confiscated amount and a small fee to the
    // caller for being first.
    if on_behalf_of_addr == fee_recipient_addr {
        on_behalf_of_yield_taken +=
-           lockup_call::confiscate(self.lockup_addr.get(), victim_addr,
on_behalf_of_addr)?;
+       confiscated;
+       erc20_call::transfer(STAKED_ARB_ADDR, on_behalf_of_addr,
confiscated)?;
    } else {
```

Mitigation

The issue was fixed as recommended.

[C-06] Existing commits will be overridden by other users

Severity

Impact: High, vote manipulation for existing users, which causes loss of funds.

Likelihood: High, it will happen without preconditions.

Description

In `contract_infra_market::predict`, users can submit their predicted outcome, which is saved in the epoch `commitments` map:

```
    pub fn predict(&mut self, trading_addr: Address, commit:
FixedBytes<32>) -> R<()> {
        assert_or!(self.enabled.get(), Error::NotEnabled);
        assert_or!(
            !self.campaign_call_deadline.get(trading_addr).is_zero(),
```

```
            Error::NotRegistered
        );
        let mut epochs = self.epochs.setter(trading_addr);
        let mut e = epochs.setter(self.cur_epochs.get(trading_addr));
        assert_or!(
            are_we_in_predicting_period(e.campaign_when_whinged.get(),
block_timestamp())?,
            Error::PredictingNotStarted
        );
        assert_or!(!commit.is_zero(), Error::NotAllowedZeroCommit);
->      e.commitments.setter(trading_addr).set(commit);
```

The issue is that this value is shared between all the users, instead of being user-specific: this means that the last user who predicts will decide the commit that ALL other users have submitted.

This causes a multitude of side effects, such as turning all winners into losers (eventually causing loss of funds due to slashing), or the impossibility of revealing at all due to the users' need to know the correct hash.

Recommendations

Consider modifying `commitments` as it should be a map of addresses to outcomes instead of being a single shared value.

Mitigation

The issue was fixed as recommended.

[C-07] Traders can extend the trading period to continue until after the winner is decided

Severity

Impact: High, traders can drain the contract because they can bet while knowing the winner outcome.

Likelihood: High, it will happen without preconditions.

Description

In `contract_trading_mint::mint_permit_E_90275_A_B` users can mint new shares, and there is a check to ensure that this can be done only during the trading period:

```
    // Ensure that we're not complete, and that the time hasn't expired.
    assert_or!(
        self.when_decided.get().is_zero()
            || self.is_shutdown.get()
->          || self.time_ending.get() > U64::from(block_timestamp()),
        Error::DoneVoting
    );
```

The issue is that all of these conditions should be true simultaneously to ensure that, but this isn't the case.

For example, a user can keep minting shares to extend the `time_ending` period by 3 hours, even after the winner is decided so that the previous assertion will always pass.

Recommendations

Consider modifying the check so that it uses `&&` instead of `||` for all conditions.

Mitigation

The issue was fixed as recommended.

[C-08] Traders can mint shares even after the result is decided

Severity

Impact: High, traders can drain the contract because they can bet while knowing the winner outcome.

Likelihood: High, it will happen without preconditions.

Description

In `contract_trading_mint::mint_permit_E_90275_A_B` users can mint new shares, and there is a check to ensure that this can be done only when the winner is not decided and the contract is enabled:

```
    // Ensure that we're not complete, and that the time hasn't expired.
    assert_or!(
        self.when_decided.get().is_zero()
->          || self.is_shutdown.get()
            || self.time_ending.get() > U64::from(block_timestamp()),
        Error::DoneVoting
    );
```

`is_shutdown` is set to true after the winner is decided. The issue is that the check is inverted, it should assert that the contract is NOT in the shutdown state (i.e. the winner is not decided yet).

Due to **C-07**, this leads to a similar impact, as users are able to drain the contract because they will know who the winner is.

If **C-07** was fixed, due to this issue it would be impossible to trade at all, as the check assumes that the contract must be in the shutdown state.

Recommendations

Consider modifying the check so that it uses `!self.is_shutdown.get()` instead of `self.is_shutdown.get()`.

Mitigation

The issue was fixed as recommended.

[H-01] Campaigns cannot be escaped after the call deadline

Severity

Impact: High, escape will be used to redeem funds, which will be impossible to do.

Likelihood: Medium, it will happen without preconditions, but should be uncommon as this means that no one must have called the campaign.

Description

In `contract_infra_market::escape`, a campaign can be escaped after the deadline call, if no one has called. The issue is that the deadline logic check is inverted, so the opposite happens.

This has two side effects:

1. The function will revert after the deadline has passed, so it will be impossible to escape.
2. Anyone can escape before the deadline, even if this shouldn't happen.

Recommendations

Consider the following fix:

```
    assert_or!(
-       self.campaign_call_deadline.get(trading_addr) >
U64::from(block_timestamp())
+       self.campaign_call_deadline.get(trading_addr) <
U64::from(block_timestamp())
            && !self.campaign_has_escaped.get(trading_addr)
            && (indeterminate_winner || is_calling_period),
        Error::CannotEscape
    );
```

Mitigation

The issue was fixed as recommended.

[H-02] Fees for the team are never collected

Severity

Impact: Medium, loss of revenue for the protocol owners.

Likelihood: High, it will happen without preconditions.

Description

In `contract_trading_mint::internal_mint`, a percentage of the total trading amount is reserved as revenue for the project owners:

```
    // Here we do some fee adjustment to send the fee recipient their
money.
    let fee_for_creator = (value * FEE_CREATOR_MINT_PCT) / FEE_SCALING;
    fusdc_call::transfer(self.fee_recipient.get(), fee_for_creator)?;
    // Collect some fees for the team (for moderation reasons for
screening).
->  let fee_for_team = (value * FEE_SPN_MINT_PCT) / FEE_SCALING;
    let value = value - fee_for_creator - fee_for_team;
```

The issue is that this amount is never tracked or collected, resulting in a loss of revenue for the protocol owners.

Recommendations

Consider collecting the funds:

```
    // Here we do some fee adjustment to send the fee recipient their
money.
    let fee_for_creator = (value * FEE_CREATOR_MINT_PCT) / FEE_SCALING;
    fusdc_call::transfer(self.fee_recipient.get(), fee_for_creator)?;
    // Collect some fees for the team (for moderation reasons for
screening).
    let fee_for_team = (value * FEE_SPN_MINT_PCT) / FEE_SCALING;
+   fusdc_call::transfer(self.owner_address.get(), fee_for_team)?;
    let value = value - fee_for_creator - fee_for_team;
```

Mitigation

The issue was fixed as recommended.

[M-01] MEV bots are incentivized to randomly `call` campaigns once they are registered

Severity

Impact: Medium, loss of yield for legit `call` users, and the system will be less efficient overall.

Likelihood: Medium, it will be profitable to do, especially for binary outcomes (50/50 of earning free funds, or gaining nothing if wrong).

Description

In `contract_infra_market::call` users will be paid if they `call` a campaign and no one whinged it.

This incentivizes users to call a campaign with a random result as fast as possible to have a chance to earn those fees, as the user doesn't lose anything other than gas fees if they are wrong.

Indeed, the result will eventually be correct due to the whinging mechanism, but this will cause strain on the system because the process will be overall longer, due to having to do the whole whinge/predict/reveal path 50% of the time (it is expected that the accuracy of legit callers would be much higher than 50% on average).

Moreover legit callers won't be able to call any campaign as it's a first come first served opportunity.

Recommendations

Consider adding a bond as a requirement for calling (similar to the whinge process), which will be refunded if the caller called the correct outcome (no one whinged it, or the whinger was wrong).

Mitigation

The issue was fixed as recommended.

[M-02] Epochs can be increased before the end of anything goes period

Severity

Impact: Medium, the epoch is increased before the intended period.

Likelihood: Medium, this can occur only on campaigns where there is an inconclusive result.

Description

In `contract_infra_market::call` users can increase the epoch on a campaign that has an inconclusive result:

```
      if campaign_winner_set
          && campaign_winner.is_zero()
          && are_we_after_anything_goes(campaign_when_whinged,
block_timestamp())?
      {
          self.cur_epochs.setter(trading_addr).set(new_epoch);
          epochs.setter(new_epoch)
      } else {
          epochs.setter(cur_epoch)
      }
```

The issue is that `timing_infra_market::are_we_after_anything_goes` tracks the start of the period instead of the end, so the epoch can be increased before the anything-goes period ends.

Recommendations

Consider the following fix in `timing_infra_market`:

```
-    pub const SIX_DAYS: u64 = 518400;
+    pub const EIGHT_DAYS: u64 = 691200;

     pub fn are_we_after_anything_goes(whinged_ts: U64, ts: u64) ->
Result<bool, Error> {
         if whinged_ts.is_zero() {
             return Err(Error::WhingedTimeUnset);
         }
-        Ok(ts > u64::from_be_bytes(whinged_ts.to_be_bytes()) + SIX_DAYS)
+        Ok(ts > u64::from_be_bytes(whinged_ts.to_be_bytes()) + EIGHT_DAYS)
     }
```

Mitigation

The issue was fixed, as the anything goes period was removed entirely.

[L-01] Contracts can't be disabled

In `contract_lockup` and `contract_infra_market` there isn't any function able to disable the contract (they are enabled by default on initialization).

Consider adding a function to do so, in case the contract has any bugs it can be stopped.

Mitigation

The issue was fixed as recommended.

[L-02] `are_we_after_whinging_period` uses a wrong parameter

In `timing_infra_market::are_we_after_whinging_period`, the function uses a `whinged_ts` parameter instead of a `called_ts` parameter. This could cause a wrong usage of this function if the caller uses the whinging timestamp instead of the calling timestamp.

The current codebase passes the correct argument when this function is used, so there isn't any further impact.

Mitigation

The issue was acknowledged.

[L-03] Inframarket periods are overlapping

In `timing_infra_market`, each period overlaps for a single block:

```
    ($func_name:ident, $when:ident, $from_days:expr, $unset_err:ident,
$days:expr) => {
        pub fn $func_name($when: U64, current_ts: u64) -> Result<bool,
Error> {
            if $when.is_zero() {
                return Err(Error::$unset_err);
            }
            let when = u64::from_be_bytes($when.to_be_bytes()) +
($from_days * 24 * 60 * 60);
            let until = when + ($days * 24 * 60 * 60);
->          Ok(current_ts >= when && current_ts <= until)
        }
    };
```

This means that it's possible, for example, to be in the reveal and the sweep period at the same time. Consider not including the end of the period in this check.

Mitigation

The issue was acknowledged.