Security Assessment Report

Monaco Protocol v0.12.0

November 14, 2023

Summary

The sec3 team (formerly Soteria) was engaged to do a thorough security analysis of the Monaco Protocol Solana smart contract at https://github.com/MonacoProtocol/protocol. The initial audit was done on the source code of the following version

- **Contract "monaco_protocol":**
  - v0.12.0, commit `e2e9c881c07c1cc98de658939fa0866e7c6141e0`

The review revealed 6 issues. The team responded with a second version for the post-audit review to check if the reported issues have bee addressed. The audit is concluded after reviewing commit `a9b4d24c4388db3a8df0af02d59cd3a737232f0f` with all fixes applied.

Table of Contents

Result Overview

In total, the audit team found the following issues.

MONACO PROTOCOL v0.12.0		
Issue	Impact	Status
[L-1] Inconsistent handling of cancelled orders	Low	Resolved
[L-2] The same position in update_product_commission_contributions	Low	Resolved
[I-1] Seed collision	Info	Resolved
[I-2] Redundant range check	Info	Resolved
[I-3] market_type_discriminator validation	Info	Resolved
[I-4] Surplus transfer when MarketStatus::ReadyToClose	Info	Resolved

Findings in Detail

[L-1] Inconsistent handling of cancelled orders

```
/* monaco_protocol/src/instructions/order/cancel_order.rs */
009 | pub fn cancel_order(ctx: Context<CancelOrder>) -> Result<()> {
033 |     ctx.accounts.order.void_stake_unmatched();
045 |     // if never matched close
046 |     if order.stake == order.voided_stake {
047 |         ctx.accounts.market.decrement_account_counts()?;
048 |         ctx.accounts
049 |             .order
050 |             .close(ctx.accounts.purchaser.to_account_info())?;
051 |     }

/* MonacoProtocol/programs/monaco_protocol/src/state/order_account.rs */
063 |
064 | pub fn void_stake_unmatched(&mut self) {
065 |     self.voided_stake = self.stake_unmatched;
066 |     self.stake_unmatched = 0_u64;
067 |     if self.order_status == OrderStatus::Open {
068 |         self.order_status = OrderStatus::Cancelled;
069 |     }
070 | }
```

For orders that are never matched (with `Open` status), `cancel_order()` will update their status to `Cancelled` in `void_stake_unmatched()`. Then, `decrement_account_counts()` will update the unsettled and unclosed account counters. Finally, these never-matched order accounts will be closed. So, the protocol doesn't have to deal with them in the settlements and market closures.

```
/* monaco_protocol/src/instructions/order/settle_order.rs */
009 | pub fn settle_order(ctx: Context<SettleOrder>) -> Result<()> {
028 |     // if never matched close
029 |     if Open.eq(&ctx.accounts.order.order_status) {
030 |         market_account.decrement_account_counts()?;
031 |         return ctx
032 |             .accounts
033 |             .order
034 |             .close(ctx.accounts.purchaser.to_account_info());
035 |     }
037 |     if ctx.accounts.order.stake_unmatched > 0_u64 {
```

```
038 |         ctx.accounts.order.void_stake_unmatched();
039 |     }
040 |     match is_winning_order(&ctx.accounts.order, market_account) {
041 |         true => ctx.accounts.order.order_status = SettledWin,
042 |         false => ctx.accounts.order.order_status = SettledLose,
043 |     };
045 |     market_account.decrement_unsettled_accounts_count()?;
```

The behavior is similar in `settle_order()`. In lines `29-34`, never-matched orders will be closed and the market unsettled/unclosed account counters will be updated.

```
/* monaco_protocol/src/instructions/order/cancel_preplay_order_post_event_start.rs */
011 | pub fn cancel_preplay_order_post_event_start(
012 |     market: &Market,
013 |     market_matching_pool: &mut MarketMatchingPool,
014 |     order: &mut Order,
015 |     market_position: &mut MarketPosition,
016 | ) -> Result<u64> {
046 |     order.void_stake_unmatched(); // <-- void needs to happen before refund calculation
047 |     let refund = market_position::update_on_order_cancellation(market_position, order)?;
049 |     Ok(refund)
050 | }
```

However, the behavior in `cancel_preplay_order_post_event_start()` is different. The status of never-matched orders will be set to `OrderStatus::Cancelled`. The accounts will not be closed and the unsettled/unclosed account counters will not be updated either.

For these unclosed orders with `Cancelled` status, as the unclosed account counter will be updated in `close_order()`, the unsettled account counter needs to be updated. `cancel_order()` cannot do that because the account doesn't satisfy the condition `order.stake_unmatched > 0_u64`. `void_order()` does not do so as the market status doesn't match (`MarketStatus::ReadyToVoid` required).

Therefore, the only option is `settle_order()`. However, the order status of these never-matched orders will become `SettledWin` or `SettledLose`, which seems inconsistent.

Resolution

This issue has been fixed by `07c6faaa`.

[L-2] The same position in update_product_commission_contributions

```
/* monaco_protocol/src/state/market_position_account.rs */
007 | pub struct MarketPosition {
008 |     pub purchaser: Pubkey,
009 |     pub market: Pubkey,
010 |     pub paid: bool,
011 |     pub market_outcome_sums: Vec<i128>,
012 |     pub unmatched_exposures: Vec<u64>,
013 |     pub payer: Pubkey, // solana account fee payer
014 |     pub matched_risk: u64,
015 |     pub matched_risk_per_product: Vec<ProductMatchedRiskAndRate>,
016 | }

/* monaco_protocol/src/instructions/matching/matching_one_to_one.rs */
014 | pub fn match_orders(ctx: &mut Context<MatchOrders>) -> Result<()> {
119 |     // update product commission tracking for matched risk
120 |     update_product_commission_contributions(market_position_for, order_for, stake_matched)?;
121 |     update_product_commission_contributions(
122 |         market_position_against,
123 |         order_against,
124 |         calculate_risk_from_stake(stake_matched, selected_price),
125 |     )?;
```

The function update_product_commission_contributions() will update the matched_risk and matched_risk_per_product fields in both market_position_for and market_position_against accounts.

However, when they are referring to the same market_position account, they should be synchronized after each update_product_commission_contributions, similar to caling copy_market_position() after each order::match_order()

Resolution

This issue has been fixed commit d2064c14 and 89e2807f.

[I-1] Seed collision

```
/* monaco_protocol/src/context.rs */
531 | #[derive(Accounts)]
532 | #[instruction(
533 |     event_account: Pubkey,
534 |     market_type_discriminator: String,
535 |     market_type_value: String,
536 | )]
537 | pub struct CreateMarket<'info> {
538 |     pub existing_market: Option<Account<'info, Market>>,
539 |
540 |     #[account(
541 |         init,
542 |         seeds = [
543 |             event_account.as_ref(),
544 |             market_type.key().as_ref(),
545 |             market_type_discriminator.as_ref(),
546 |             b"-".as_ref(),
547 |             market_type_value.as_ref(),
548 |             b"-".as_ref(),
549 |             get_create_market_version(&existing_market).to_string().as_ref(),
550 |             mint.key().as_ref(),
551 |         ],
552 |         bump,
553 |         payer = market_operator,
554 |         space = Market::SIZE
555 |     )]
556 |     pub market: Box<Account<'info, Market>>,
```

The seeds of the market account are expanded with [..., market_type_discriminator.as_ref(), b"-".as_ref(), market_type_value.as_ref(), ...,]. However, market_type_discriminator and market_type_value are unvalidated strings. If one contains -, different combinations may lead to the same seeds. E.g (market_type_discriminator = "abc-efg", market_type_value = "123") and (market_type_discriminator = "abc", market_type_value = "efg-123")

However, besides panics, this doesn't seem to have a bigger impact.

Resolution

This issue has been fixed by a34ec1ee.

[I-2] Redundant range check

```
/* monaco_protocol/src/instructions/market/create_market.rs */
046 | require!(
047 |     ctx.accounts.mint.decimals >= PRICE_SCALE,
048 |     CoreError::MintDecimalsUnsupported
049 | );
050 | let decimal_limit = ctx.accounts.mint.decimals.saturating_sub(max_decimals);
051 | require!(PRICE_SCALE <= decimal_limit, CoreError::MaxDecimalsTooLarge);
```

`ctx.accounts.mint.decimals.saturating_sub(max_decimals) >= PRICE_SCALE` implies `ctx.accounts.mint.decimals >= PRICE_SCALE`, given `max_decimals` is a `u8` integer.

Resolution

The team acknowledged this issue and decided to keep it.

[I-3] market_type_discriminator validation

```
/* monaco_protocol/src/instructions/market/create_market.rs */
053 | require!(
054 |     ctx.accounts.market_type.requires_discriminator != market_type_discriminator.is_empty(),
055 |     CoreError::MarketTypeDiscriminatorUsageIncorrect
056 | );
057 | require!(
058 |     ctx.accounts.market_type.requires_value != market_type_value.is_empty(),
059 |     CoreError::MarketTypeValueUsageIncorrect
060 | );
```

When `requires_discriminator` is `false`, user-provided `market_type_discriminator` must be empty. Should they be optional instead when it's not required?

The validation logic for `market_type_value` is similar.

Resolution

This issue is fixed by commit `a9b4d24c`.

Based on the test "failure when market type discriminator contains the seed separator", the following seems to be the expected behaviors.

- when `requires_xxx` is `true`, `market_type_xxx` should be `Some(a non-empty string)`
- otherwise, `market_type_xxx` should be None

With the fix, when `requires_xxx` is `true`, `market_type_xxx` can be `Some("")`, which could lead to the same result as `requires_xxx = false, market_type_xxx = None` due to `.unwrap_or(&"".to_string())` in the PDA seeds.

However, the collision here won't make a difference in the PDA seeds of `market`, as there cannot be two `market_type` PDAs with the same name but different `requires_xxx`.

It may be a good idea to further clarify the expected behavior just in case the collision may lead to different behaviors in the future.

```
if (requires_discriminator) {
    require!(
        market_type_discriminator.is_some()
            && !market_type_discriminator.as_ref().unwrap().is_empty(),
        CoreError::MarketTypeDiscriminatorUsageIncorrect
    );
} else {
    require!(
        market_type_discriminator.is_none(),
        CoreError::MarketTypeDiscriminatorUsageIncorrect
    );
}
```

[I-4] Surplus transfer when MarketStatus::ReadyToClose

market_escrow.amount == 0_u64 is the pre-condition before the status transition to ReadyToClose. No need to transfer because there is no surplus for MarketStatus::ReadyToClose?

```
/* monaco_protocol/src/instructions/market/update_market_status.rs */
106 | pub fn ready_to_close(market: &mut Market, market_escrow: &TokenAccount) -> Result<()> {
107 |     require!(
108 |         Settled.eq(&market.market_status) || Voided.eq(&market.market_status),
109 |         CoreError::MarketNotSettledOrVoided
110 |     );
112 |     require!(
113 |         market_escrow.amount == 0_u64,
114 |         CoreError::SettlementMarketEscrowNonZero
115 |     );
117 |     market.market_status = ReadyToClose;
118 |     Ok(())
119 | }

/* monaco_protocol/src/instructions/market/escrow.rs */
012 | const TRANSFER_SURPLUS_ALLOWED_STATUSES: [MarketStatus; 3] = [
013 |     MarketStatus::Settled,
014 |     MarketStatus::ReadyToClose,
015 |     MarketStatus::Voided,
016 | ];
017 |
018 | pub fn transfer_market_escrow_surplus<'info>(
019 |     market: &Account<'info, Market>,
020 |     market_escrow: &Account<'info, TokenAccount>,
021 |     destination: &Account<'info, TokenAccount>,
022 |     token_program: &Program<'info, Token>,
023 | ) -> Result<()> {
024 |     require!(
025 |         TRANSFER_SURPLUS_ALLOWED_STATUSES.contains(&market.market_status),
026 |         CoreError::MarketInvalidStatus
027 |     );
028 |     transfer::transfer_market_escrow_surplus(market_escrow, destination, token_program, market)
029 | }
```

Resolution

This issue has been fixed by a8799b5b.

Appendix: Methodology and Scope of Work

The sec3 (formerly Soteria) audit team, which consists of Computer Science professors and industrial researchers with extensive experience in Solana smart contract security, program analysis, testing, and formal verification, performed a comprehensive manual code review, software static analysis, and penetration testing.

Assisted by the sec3 Scanner developed in-house, the audit team particularly focused on the following work items:

- Check common security issues.
    - Missing ownership checks
    - Missing signer checks
    - Signed invocation of unverified programs
    - Solana account confusions
    - Arithmetic over- or underflows
    - Numerical precision errors
    - Loss of precision in calculation
    - Insufficient SPL-Token account verification
    - Missing rent exemption assertion
    - Casting truncation
    - Did not follow security best practices
    - Outdated dependencies
    - Redundant code
    - Unsafe Rust code
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of the scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a sec3 (the "Company") and BetDEX Labs (the "Client"). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

Founded by leading academics in the field of software security and senior industrial veterans, sec3 (formerly Soteria) is a leading blockchain security company that currently focuses on Solana programs. We are also building sophisticated security tools that incorporate static analysis, penetration testing, and formal verification.

At sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.

sec3™