Sec3™

Security Assessment Report

Monaco Protocol v0.14.0

April 11, 2024

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Monaco Protocol v0.14.0 smart contracts.

The artifact of the audit was the source code of the following programs, excluding tests, in https://github.com/MonacoProtocol/protocol.

The initial audit focused on the following versions and revealed 9 issues or questions.

program	type	commit
monaco_protocol	solana	20365bb5feefef167e824f06279dd8d79408e040

The post-audit review was conducted on the following version to check if the reported issues had been addressed.

program	type	commit
monaco_protocol	solana	500c2a04b109a79ab3cd7110e4c695cd98bf0716

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
MONACO_PROTOCOL		
[H-01] Missing market_outcome_index and for_outcome checks	High	Resolved
[L-01] DoS in create_order_request	Low	Resolved
[L-02] Missing length checks for liquidity vectors	Low	Resolved
[L-03] Anyone can create order requests without paying	Low	Resolved
[I-01] Inconsistent behaviors when cancel never matched orders	Info	Resolved
[I-02] Missing outcome title length checks	Info	Resolved
[I-03] Check event_start_timestamp against market_lock_timestamp	Info	Resolved
[I-04] Risk of not recording all matched product risks	Info	Resolved
[I-05] Missing data.price checks	Info	Resolved

Findings in Detail

MONACO_PROTOCOL

[H-01] Missing market_outcome_index and for_outcome checks

The "`process_order_match`" instruction matches orders from the "`market_matching_queue`" and "`market_matching_pool`".

```
/* programs/monaco_protocol/src/lib.rs */
pub fn process_order_match(ctx: Context<ProcessOrderMatch>) -> Result<()> {
    let stake_unmatched = ctx.accounts.maker_order.stake_unmatched;
    if stake_unmatched == 0 {
    } else {
        let refund_amount = instructions::matching::on_order_match(
            &mut ctx.accounts.market_matching_queue,
            &mut ctx.accounts.market_matching_pool,
            &ctx.accounts.maker_order.key(),
            &mut ctx.accounts.maker_order,
        )?;

/* programs/monaco_protocol/src/instructions/matching/on_order_match.rs */
pub fn on_order_match(
    market_matching_queue: &mut MarketMatchingQueue,
    market_matching_pool: &mut MarketMatchingPool,
    maker_order_pk: &Pubkey,
    maker_order: &mut Order,
) -> Result<u64> {
    match market_matching_queue.matches.peek_mut() {
        Some(taker_order) => {
            // determine matched stake
            let stake = maker_order.stake_unmatched.min(taker_order.stake);
```

However, it doesn't check if the orders to be matched have the same "`market_outcome_index`" and they have different "`for_outcome`".

```
/* programs/monaco_protocol/src/context.rs */
pub struct ProcessOrderMatch<'info> {
    #[account(
        has_one = market @ CoreError::MatchingMarketMismatch,
    )]
    pub market_matching_pool: Box<Account<'info, MarketMatchingPool>>,
    #[account(
        has_one = market @ CoreError::MatchingMarketMismatch,
    )]
    pub market_matching_queue: Box<Account<'info, MarketMatchingQueue>>,
    #[account(
0549 |         has_one = market @ CoreError::MatchingMarketMismatch,
0550 |         constraint = *market_matching_pool.orders.peek(0)
0551 |             .ok_or(CoreError::MatchingQueueIsEmpty)? == maker_order.key() @ ...,
0552 |     )]
0553 |     pub maker_order: Account<'info, Order>,

0122 | pub struct ProcessOrderRequest<'info> {
0160 |     #[account(
0161 |         init_if_needed,
0162 |         seeds = [
0164 |             order_request_queue.order_requests
0165 |                 .peek_front()
0166 |                 .ok_or(CoreError::OrderRequestQueueIsEmpty)?
0167 |                 .market_outcome_index.to_string().as_ref(),
0179 |         ],
0183 |     )]
0184 |     pub market_matching_pool: Box<Account<'info, MarketMatchingPool>>,

1035 | pub struct OpenMarket<'info> {
1050 |     #[account(
1051 |         init,
1052 |         seeds = [
1053 |             b"matching".as_ref(),
1054 |             market.key().as_ref(),
1055 |         ],
1059 |     )]
1060 |     pub matching_queue: Account<'info, MarketMatchingQueue>,
```

In particular, “`maker_order`” is from “`market_matching_pool`”, which has “`market_outcome_index`” in its PDA seeds (context.rs:167). However, “`market_matching_queue`” is shared by all orders for a given market (context.rs:1054).

Therefore, the “`market_outcome_index`” of the “`taker_order`” from “`market_matching_queue`” (on_order_match.rs:34) and the “`maker_order`” can be different.

PoC

Order mismatches can lead to payment issues at settlement. Consider a scenario with three participants: A, B, and C.

Initially, A’s order should have been successfully matched with B’s. However, an attacker, C, subsequently forces a match with B using “`processOrderMatch`”.

If C wins, the market may not have enough funds to pay him, as he was not part of the original match and should only have received his principal back. Winning the match entitles C to a payout he was not due, leading to insufficient funds in the market. This results in B being unable to withdraw his earnings.

The core issue is that orders with mismatched outcomes are mistakenly deemed successfully matched, compromising the market's integrity in fund management.

```
/* tests/order/matching_orders_01.ts */
it("matching: market incorrect outcome", async () => {
  const stake = 10;
  const price = 3.0;
  const outcome = 0;
  const outcome_1 = 1;
  const startBalance = 100.0;
  // purchaserC as an attacker
  const [purchaserA, purchaserB, purchaserC, market] = await Promise.all([
    createWalletWithBalance(monaco.provider),
    createWalletWithBalance(monaco.provider),
    createWalletWithBalance(monaco.provider),
    monaco.create3WayMarket([price]),
  ]);
  await Promise.all([
    market.airdrop(purchaserA, startBalance),
    market.airdrop(purchaserB, startBalance),
    market.airdrop(purchaserC, startBalance),
  ]);
  // 1. create A for (0,10) B against (0,10)
  await market.forOrder(outcome, stake, price, purchaserA);
  await market.againstOrder(outcome, stake, price, purchaserB);
  // 2. create C for (1,10)
  await market.forOrder(outcome_1, stake, price, purchaserC);
  // 3. match C for (1,10) with B against (0,10)
  await market.processMatchingQueueAttack();
  // 4. EscrowBalance unable to cover total reward when outcome 1 wins
  assert.deepEqual(
    await Promise.all([
      market.getEscrowBalance(),
      market.getTokenBalance(purchaserA),
      market.getTokenBalance(purchaserB),
    ]),
    [40, 90, 80],
  );
  // settle for outcome_1
  await market.settle(outcome_1);
  // settle A,C
  await market.settleMarketPositionForPurchaser(purchaserA.publicKey);
  await market.settleMarketPositionForPurchaser(purchaserC.publicKey);
  try {
    await market.settleMarketPositionForPurchaser(purchaserB.publicKey);
    assert.fail("settle market position should have failed");
  } catch (e) {}
  });

/* tests/util/wrappers.ts */
const matchingPools =
  this.matchingPools[takerOrder.outcomeIndex + 1][takerOrder.price]; // changed
console.log("matchingPools", matchingPools);
const matchingPoolPk = takerOrder.forOutcome
  ? matchingPools.against
  : matchingPools.forOutcome;

  Order Matching Market State
      matching: market incorrect outcome (16483 ms)
```

Recommendation

Check "`taker_order`" and "`maker_order`" share the same "`market_outcome_index`" and they are a pair of for/against orders.

Resolution

This issue has been resolved by commit 40a2f810.

MONACO_PROTOCOL

[L-01] DoS in create_order_request

When the "order_request_queue" has multiple "order_request" accounts with the same "purchaser" and "distinct_seed", the "process_order_request" instruction will get stuck because it cannot dequeue the "order_request" from the "order_request_queue", due to PDA collisions when creating the corresponding "order" accounts.

In particular, in the "create_order_request" instruction, "reserved_order" accounts are initialized and then closed before exiting the instruction handler, as indicated at "context.rs:34" and "lib.rs:98", respectively.

```
/* programs/monaco_protocol/src/context.rs */
022 | pub struct CreateOrderRequest<'info> {
023 |     #[account(
024 |         init,
025 |         seeds = [
026 |             market.key().as_ref(),
027 |             purchaser.key().as_ref(),
028 |             &data.distinct_seed,
029 |         ],
033 |     )]
034 |     pub reserved_order: Account<'info, ReservedOrder>,
035 |     #[account(
036 |         mut,
037 |         seeds = [b"order_request".as_ref(), market.key().as_ref()],
038 |         bump,
039 |     )]
040 |     pub order_request_queue: Account<'info, MarketOrderRequestQueue>,

/* programs/monaco_protocol/src/lib.rs */
039 | pub fn create_order_request(
040 |     ctx: Context<CreateOrderRequest>,
041 |     data: OrderRequestData,
042 | ) -> Result<()> {
043 |     let payment = instructions::order_request::create_order_request(
052 |         &mut ctx.accounts.order_request_queue,
053 |         data,
054 |     )?;
096 |     ctx.accounts
097 |         .reserved_order
098 |         .close(ctx.accounts.payer.to_account_info())
099 | }
```

This process is designed to check for the presence of existing "order" accounts that share identical PDA seeds to prevent PDA collisions later in the "process_order_request" instruction.

```
/* programs/monaco_protocol/src/context.rs */
pub struct ProcessOrderRequest<'info> {
    #[account(
        init,
        seeds = [
            market.key().as_ref(),
            order_request_queue.order_requests
                .peek_front()
                .ok_or(CoreError::OrderRequestQueueIsEmpty)?
                .purchaser.as_ref(),
            &order_request_queue.order_requests
                .peek_front()
                .ok_or(CoreError::OrderRequestQueueIsEmpty)?
                .distinct_seed,
        ],
    )]
    pub order: Account<'info, Order>,
```

However, when there are no such orders, it's still possible to enqueue multiple "order_request" accounts with the same "purchaser" and "distinct_seed", as long as the "market_outcome_index", "for_outcome", "stake" or "expected_price" are different.

```
/* programs/monaco_protocol/src/instructions/order_request/create_order_request.rs */
pub fn create_order_request(
) -> Result<u64> {
    require!(
        !order_request_queue.order_requests.contains(&order_request),
        CoreError::OrderRequestCreationDuplicateRequest
    );
    order_request_queue
        .order_requests
        .enqueue(order_request)
        .ok_or(CoreError::OrderRequestCreationQueueFull)?;

/* programs/monaco_protocol/src/state/market_order_request_queue.rs */
impl PartialEq for OrderRequest {
    fn eq(&self, other: &Self) -> bool {
        self.market_outcome_index == other.market_outcome_index
            && self.for_outcome == other.for_outcome
            && self.stake == other.stake
            && self.expected_price == other.expected_price
            && self.distinct_seed == other.distinct_seed
            && self.purchaser == other.purchaser
    }
}

```

When the "process_order_request" instruction attempts to dequeue these "order_request" accounts from the "order_request_queue", it always fails due to PDA conflicts.

To recover, the market authority must manually intervene by executing the "`dequeue_order_request`" instruction to eliminate the problematic "`order_request`" account.

Resolution

This issue has been resolved by commit f1d27cd7.

MONACO_PROTOCOL

[L-02] Missing length checks for liquidity vectors

The size of the allocated space for “liquidities” is determined by “MarketLiquidities::SIZE”.

```
/* programs/monaco_protocol/src/context.rs */
pub struct OpenMarket<'info> {
    #[account(
        init,
        space = MarketLiquidities::SIZE
    )]
    pub liquidities: Account<'info, MarketLiquidities>,
```

The “liquidities” has two vectors: “liquidities_for” and “liquidities_against”. They are each supposed to have a maximum of “LIQUIDITIES_VEC_LENGTH” items.

However, when inserting new elements, it does not check whether the vectors have already reached their capacity, leading to space allocation errors.

```
/* programs/monaco_protocol/src/state/market_liquidities.rs */
pub struct MarketLiquidities {
    pub market: Pubkey,
    pub liquidities_for: Vec<MarketOutcomePriceLiquidity>,
    pub liquidities_against: Vec<MarketOutcomePriceLiquidity>,
}
impl MarketLiquidities {
    const LIQUIDITIES_VEC_LENGTH: usize = 30_usize;
    pub const SIZE: usize = DISCRIMINATOR_SIZE
        + PUB_KEY_SIZE // market
        + vec_size(MarketOutcomePriceLiquidity::SIZE, MarketLiquidities::LIQUIDITIES_VEC_LENGTH)
        + vec_size(MarketOutcomePriceLiquidity::SIZE, MarketLiquidities::LIQUIDITIES_VEC_LENGTH);

fn add_liquidity(
    ) -> Result<()> {
        match liquidities.binary_search_by(search_function) {
            Ok(index) => {
            }
            Err(index) => liquidities.insert(
            )
```

Resolution

This issue has been resolved by commit 0fc61ff9.

MONACO_PROTOCOL

[L-03] Anyone can create order requests without paying

In "`create_order_request`", the payment will be made from the "`purchaser_token`" token account.

```
/* programs/monaco_protocol/src/context.rs */
022 | pub struct CreateOrderRequest<'info> {
055 |     pub purchaser: Signer<'info>,
056 |     #[account(
057 |         mut,
058 |         token::mint = market.mint_account,
059 |     )]
060 |     pub purchaser_token: Account<'info, TokenAccount>,

/* programs/monaco_protocol/src/lib.rs */
039 | pub fn create_order_request(
040 |     ctx: Context<CreateOrderRequest>,
041 |     data: OrderRequestData,
042 | ) -> Result<()> {
043 |     let payment = instructions::order_request::create_order_request(
054 |     )?;
057 |     if ctx.accounts.purchaser_token.owner == ctx.accounts.purchaser_token.key() {
058 |         // Verify PDA is the correct account
059 |         let market = ctx.accounts.market.key();
060 |         Pubkey::create_program_address(
061 |             &[
062 |                 b"funding",
063 |                 market.key().as_ref(),
064 |                 &[ctx.accounts.market.funding_account_bump],
065 |             ],
066 |             &monaco_protocol::ID,
067 |         )
068 |         .map_or_else(
069 |             |_| Err(CoreError::OrderRequestCreationInvalidPayerTokenAccount.into()),
070 |             |pk| {
071 |                 require!(
072 |                     pk == ctx.accounts.purchaser_token.key(),
073 |                     CoreError::OrderRequestCreationInvalidPayerTokenAccount
074 |                 );
075 |                 Ok(())
076 |             },
077 |         )?;
079 |         transfer::funding_account_order_creation_payment(
080 |             &ctx.accounts.market_escrow,
081 |             &ctx.accounts.purchaser_token,
082 |             &ctx.accounts.token_program,
083 |             &ctx.accounts.market.key(),
084 |             ctx.accounts.market.funding_account_bump,
085 |             payment,
086 |         )?;
```

When the owner of the "`purchaser_token`" is the same account (as referenced in lib.rs:57) and the "purchaser_token" is identified as the market "funding" PDA (as outlined in lib.rs:60-77), purchasers can cover the fees using the market's "funding" account without personal payments.

```
/* programs/monaco_protocol/src/context.rs */
pub struct CreateMarket<'info> {
    #[account(
        init,
        seeds = [
            b"funding".as_ref(),
            market.key().as_ref(),
        ],
        bump,
        payer = market_operator,
        token::mint = mint,
        token::authority = funding
    )]
    pub funding: Box<Account<'info, TokenAccount>>,
```

However, since the market funding PDA is publicly known, any purchaser could bypass the payment by specifying the market "funding" account as the "purchaser_token" account.

Resolution

The team clarified that this is an intended behavior, even though the funding account is publicly known and anyone can grab it.

However, at any given time between transaction that account's balance is 0. It's only being topped up during the same transaction in which it's being drawn down. A proxy program tops the funding account and calls order request creation via CPI in the same transaction. That means if the transaction succeeds balance of funding account is consumed immediately and if it fails balance of funding account was never increased.

MONACO_PROTOCOL

[I-01] Inconsistent behaviors when cancel never matched orders

Instruction "`cancel_order`" closes order accounts if they have never been matched.

```
/* programs/monaco_protocol/src/instructions/order/cancel_order.rs */
pub fn cancel_order(ctx: Context<CancelOrder>) -> Result<()> {
    // if never matched close
    if order.stake == order.voided_stake {
        market.decrement_account_counts()?;
        ctx.accounts
            .order
            .close(ctx.accounts.payer.to_account_info())?;
    }
```

However, in "`cancel_order_post_market_lock`" and "`cancel_preplay_order_post_event_start`", never matched order accounts are marked as settled (by the counter) and not closed. Consider closing such orders too.

```
/* programs/monaco_protocol/src/instructions/order/cancel_order_post_market_lock.rs */
pub fn cancel_order_post_market_lock(
) -> Result<u64> {
    order.void_stake_unmatched(); // <-- void needs to happen before refund calculation
    if order.order_status == OrderStatus::Cancelled {
        market.decrement_unsettled_accounts_count()?;
    }


/* programs/monaco_protocol/src/instructions/order/cancel_preplay_order_post_event_start.rs */
pub fn cancel_preplay_order_post_event_start(
) -> Result<u64> {
    order.void_stake_unmatched(); // <-- void needs to happen before refund calculation
    // if never matched
    if order.stake == order.voided_stake {
        // no more settlement needed
        market.decrement_unsettled_accounts_count()?;
    }
```

Resolution

The team clarified that this is the intended behavior for now. They may reevaluate when to clean up the canceled orders in the future.

MONACO_PROTOCOL

[I-02] Missing outcome title length checks

The length of "`outcome.title`" should be less than "`MarketOutcome::TITLE_MAX_LENGTH`".

```
/* programs/monaco_protocol/src/lib.rs */
466 | pub fn initialize_market_outcome(
467 |     ctx: Context<InitializeMarketOutcome>,
468 |     title: String,
469 | ) -> Result<()> {
480 |     instructions::market::initialize_outcome(ctx, title)?;
483 | }

/* programs/monaco_protocol/src/instructions/market/create_market.rs */
150 | pub fn initialize_outcome(ctx: Context<InitializeMarketOutcome>, title: String) -> Result<()> {
158 |     ctx.accounts.outcome.title = title;

/* programs/monaco_protocol/src/state/market_outcome_account.rs */
016 | impl MarketOutcome {
017 |     pub const TITLE_MAX_LENGTH: usize = 100;
```

Resolution

This issue has been resolved by commit 547cd4bd.

MONACO_PROTOCOL

[I-03] Check event_start_timestamp against market_lock_timestamp

In “`update_market_event_start_time_internal()`”, when “`market.inplay_enabled`” is “`false`”, the incoming “`event_start_time`” should not be earlier than the “`market.market_lock_timestamp`”.

```rust
/* monaco_protocol/src/instructions/market/update_market_event_start_time.rs */
fn update_market_event_start_time_internal(
    market: &mut Market,
    event_start_time: i64,
    now: i64,
) -> Result<()> {
    // market event start time cannot be change after market moves to inplay
    require!(!market.is_inplay(), CoreError::MarketAlreadyInplay);

    if event_start_time < now {
        msg!(
            "Update Market: event start time {} must not be in the past.",
            event_start_time.to_string()
        );
        return Err(error!(CoreError::MarketEventStartTimeNotInTheFuture));
    }

    market.event_start_timestamp = event_start_time;

    Ok(())
}
```

Resolution

The team clarified that this is intended. Market operators should manage them.

MONACO_PROTOCOL

[I-04] Risk of not recording all matched product risks

In "update_product_commission_contributions()", the "matched_risk_per_product" of the market position is updated based on an incoming order.

```
/* monaco_protocol/src/instructions/market_position/update_product_commission_contributions.rs */
006 | pub fn update_product_commission_contributions(
010 | ) -> Result<()> {
025 |     match matched_risk_per_product
026 |         .iter_mut()
027 |         .find(|p| p.product == order_product.unwrap() && p.rate == order_product_commission_rate)
028 |     {
029 |         Some(product_matched_risk_and_rate) => {
030 |             product_matched_risk_and_rate.risk = product_matched_risk_and_rate
031 |                 .risk
032 |                 .checked_add(risk_matched)
033 |                 .unwrap();
034 |         }
035 |         None => {
036 |             if matched_risk_per_product.len() < ProductMatchedRiskAndRate::MAX_LENGTH {
037 |                 matched_risk_per_product.push(ProductMatchedRiskAndRate {
038 |                     product: order_product.unwrap(),
039 |                     rate: order_product_commission_rate,
040 |                     risk: risk_matched,
041 |                 });
042 |             }
043 |         }
044 |     }
047 | }
```

If the order's "product" and "product_commission_rate" are not present in the matched risk list, a new "ProductMatchedRiskAndRate" object is created and added to the "matched_risk_per_product" list (refer to lines 35-43).

However, if the "matched_risk_per_product" has reached its maximum capacity (as mentioned in line 36), the matched risk associated with a new product will not be recorded.

```
/* programs/monaco_protocol/src/instructions/market_position/settle_market_position.rs */
084 | fn calculate_product_commission_payments(
089 | ) -> (u64, Vec<PaymentInfo>) {
097 |     for product_risk_and_rate in &market_position.matched_risk_per_product {
098 |         let product_commission_at_rate = calculate_commission_for_risk_at_rate(
099 |             protocol_commission_rate,
100 |             market_position.matched_risk,
101 |             position_profit,
            product_risk_and_rate,
        );

        payments.push(PaymentInfo {
            to: product_risk_and_rate.product,
            from: market_escrow,
            amount: product_commission_at_rate,
        });
}
```

Subsequently, when calculating product commissions, payments are generated based on the “matched_risk_per_product” of the market position (referenced in settle_market_position.rs:97). Consequently, payments for any matched risks that were not recorded due to capacity constraints will not be processed.

Resolution

The team clarify that this is intentional.

The issue is that “update_product_commission_contributions” is called during a matching process and if they raised that error there would be no simple way of handling it. It would block our cranks and to restore them we would have to cancel the order out somehow, which would also be not very transparent to an order owner.

They have simply decided that if a given user for a given market somehow manages to use 20 different products (which is unlikely since there is pretty much only 1 atm) she simply won’t have to pay for them. Also, the product is an optional value so users can bypass it by interacting directly with the protocol if they wish.

MONACO_PROTOCOL

[I-05] Missing data.price checks

When creating order requests, if "price_ladder_account.prices" is empty (referenced in create_order_request.rs:123), "data.price" is not validated.

```
/* programs/monaco_protocol/src/lib.rs */
039 | pub fn create_order_request(
040 |     ctx: Context<CreateOrderRequest>,
041 |     data: OrderRequestData,
042 | ) -> Result<()> {
043 |     let payment = instructions::order_request::create_order_request(
051 |         &ctx.accounts.price_ladder,
053 |         data,
054 |     )?;

/* programs/monaco_protocol/src/instructions/order_request/create_order_request.rs */
015 | pub fn create_order_request(
023 |     price_ladder: &Option<Account<PriceLadder>>,
025 |     data: OrderRequestData,
026 | ) -> Result<u64> {
027 |     let now: UnixTimestamp = current_timestamp();
028 |     validate_order_request(market, market_outcome, price_ladder, &data, now)?;

/* programs/monaco_protocol/src/instructions/order_request/create_order_request.rs */
100 | fn validate_order_request(
103 |     price_ladder: &Option<Account<PriceLadder>>,
104 |     data: &OrderRequestData,
106 | ) -> Result<()> {
119 |     if market_outcome.price_ladder.is_empty() {
120 |         // No prices included on the outcome, use a PriceLadder or default prices
121 |         match price_ladder {
122 |             Some(price_ladder_account) => require!(
123 |                 price_ladder_account.prices.is_empty()
124 |                     || price_ladder_account.prices.contains(&data.price),
125 |                 CoreError::CreationInvalidPrice
126 |             ),
```

Consider validating "data.price" by invoking "validate_prices".

Resolution

This issue was resolved by commit f5a3892.

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 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.
- 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 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 C̀lient''). 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. 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™