Competitive Security Assessment

DomiChain_AI

Aug 30th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.
The comprehensive examination and auditing scope includes:
• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
• Verify the code base is compliant with the most up-to-date industry standards and security best practices.
• Comprehensive line-by-line manual code review of the entire codebase by industry experts.
The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

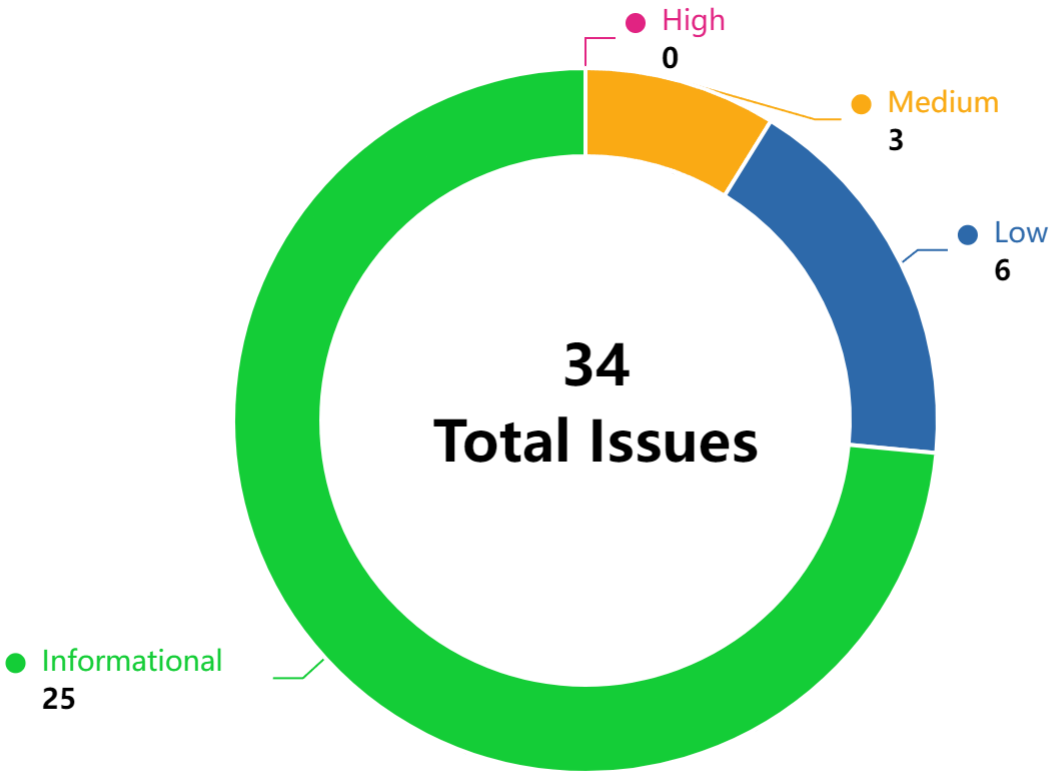
Overview

Project Name	DomiChain_AI
Language	solidity
Codebase	• https://github.com/Domino-Blockchain/AI-proxy https://github.com/Domino-Blockchain/AI-node • audit version - e4a92faadf8f3bcdd55ec4912285c1b867567f52db8f3629df519b302f88cb6f8676b26cc20b5cd1 • final version - 3151420f6d05fc5cee5f7edac891475b1336e1de

Audit Scope

File	SHA256 Hash
https://github.com/Domino-Blockchain/AI-proxy	e4a92faadf8f3bcdd55ec4912285c1b867567f52
https://github.com/Domino-Blockchain/AI-node	db8f3629df519b302f88cb6f8676b26cc20b5cd1

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
DMC-1	The `transaction_handler` function will panic when it fails to decode `Transaction`	Logical	Medium	Fixed	***
DMC-2	SQL Injection Risk	Logical	Medium	Fixed	***
DMC-3	Anyone can potentially bloat the system to hike up the protocol's computational cost due to the way fees are charged	Logical	Medium	Fixed	***
DMC-4	The function `get_risk_score_by_timestamp` can query datas which `timestamp` are greater than the current time.	Logical	Low	Fixed	***
DMC-5	Missing exception handling	DOS	Low	Acknowledged	***
DMC-6	Lack of Rate Limiting in Proxy Handler	DOS	Low	Fixed	***

DMC-7	Incorrect http request method	Logical	Low	Fixed	***
DMC-8	Http response not check	Logical	Low	Acknowledged	***
DMC-9	Computational transaction with `ai_node_manager#ai_node_manage()` could cause for this function to not work if protocol heaavily scales	Logical	Low	Mitigated	***
DMC-10	When there is too much data in the database, the http request may timeout	Logical	Informational	Acknowledged	***
DMC-11	Using `unwrap()` could lead to the application to panic	Logical	Informational	Fixed	***
DMC-12	Use of requests without the timeout argument	Logical	Informational	Acknowledged	***
DMC-13	The use of `.get()` to read array data can lead to out-of-bounds access	Logical	Informational	Fixed	***
DMC-14	The Rust contracts are using outdated crates with known vulnerabilities	Logical	Informational	Acknowledged	***
DMC-15	Sensitive Data Exposure	Write to Arbitrary Storage Location	Informational	Fixed	***
DMC-16	Return value is never checked	Logical	Informational	Acknowledged	***
DMC-17	Protocol currently hardciodes the mspc channel capacity	Logical	Informational	Acknowledged	***
DMC-18	Predictable Random Values: Potential Duplicate Values in Repeated Runs	Weak Sources of Randomness	Informational	Acknowledged	***
DMC-19	Potential Race Condition in `ds_model_predict` and `dos_model_predict`	Logical	Informational	Acknowledged	***

DMC-20	Potential Denial of Service (DoS) Attack in AI-proxy/src/main.rs	DOS	Informational	Fixed	***
DMC-21	Optimizing Resource Usage When Processing Fewer Transactions	Logical	Informational	Fixed	***
DMC-22	Missing `Pool::disconnect` in `main` Function	Logical	Informational	Fixed	***
DMC-23	Missing Event Emissions can lead to No Transparency and Traceability in case of Security Incident	Language Specific	Informational	Fixed	***
DMC-24	Lack of authentication for http server	DOS	Informational	Acknowledged	***
DMC-25	It is recommended to add a request limiter to the `predict` function	Logical	Informational	Acknowledged	***
DMC-26	Insecure Storage of Database Credentials	Logical	Informational	Fixed	***
DMC-27	Insecure Node Registration over HTTP	Logical	Informational	Fixed	***
DMC-28	Incorrect Timestamp Comparison Logic Leads to Improper Data Inclusion	Logical	Informational	Fixed	***
DMC-29	Inadequate Handling of Concurrency	Race condition	Informational	Fixed	***
DMC-30	Improper Input Validation in Command-Line Argument Parsing	Race condition	Informational	Fixed	***
DMC-31	If there are no active nodes, all transactions are silently ignored	Logical	Informational	Acknowledged	***
DMC-32	Cursor not properly closed in database operation	Logical	Informational	Fixed	***

DMC-33	Cloning using arc multiple times in the loop in `ai_node_manage()` unnecessarily hikes up cost	Code Style	Informational	Fixed	***
DMC-34	Base64 malleable risk	Language Specific	Informational	Acknowledged	***

DMC-1:The `transaction_handler` function will panic when it fails to decode `Transaction`

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/AI-proxy/src/ai_handler.rs#L59-63

```
59: fn from(s: &str) -> Self {
60:                 match s {
61:                     "base58" => Self::Base58,
62:                     "base64" => Self::Base64,
63:                     _ => panic!("Unsupported encoding"),
```

- code/AI-proxy/src/ai_handler.rs#L80-L131

```
80: if request.method == "sendTransaction" {
81:             let params_array = request.params.unwrap();
82:             let tx_str = params_array
83:                 .get(0)
84:                 .unwrap_or_else(|| panic!("No transaction in params"))
85:                 .as_str()
86:                 .unwrap();
87:
88:             let tx_options = params_array.get(1);
89:             let tx_encoding = match tx_options {
90:                 Some(options) => match options.get("encoding") {
91:                     Some(encoding) => TransactionEncoding::from(encoding.as_str().unwrap()),
92:                     None => TransactionEncoding::Base58,
93:                 },
94:                 None => TransactionEncoding::Base58,
95:             };
96:
97:             let decoded_tx_bytes = match tx_encoding {
98:                 TransactionEncoding::Base58 => match bs58::decode(tx_str).into_vec() {
99:                     Ok(decoded_bytes) => Ok(decoded_bytes),
```

```
100:                     Err(err) => Err(format!("Error decoding base58: {err:?}")),
101:                 },
102:                 TransactionEncoding::Base64 => match base64::decode(tx_str) {
103:                     Ok(decoded_bytes) => Ok(decoded_bytes),
104:                     Err(err) => Err(format!("Error decoding base64: {err:?}")),
105:                 },
106:             };
107:
108:             let tx: Transaction = match decoded_tx_bytes {
109:                 Ok(decoded_bytes) => {
110:                     bincode::deserialize(&decoded_bytes).unwrap_or_else(|err| {
111:                         error!("Error deserializing transaction: {err:?}");
112:                         // Provide a default value or handle the error case accordingly
113:                         Default::default()
114:                     })
115:                 }
116:                 Err(err) => {
117:                     error!("{err}");
118:                     // Provide a default value or handle the error case accordingly
119:                     Default::default()
```

```
120:                 }
121:             };
122:
123:             let tx_message = tx.message();
124:             let num_required_signatures = tx_message.header.num_required_signatures;
125:
126:             const BASE_CONGESTION: f32 = 5_000.0;
127:             const RATE: f32 = 1_000_000_000.0;
128:             let fee = BASE_CONGESTION * num_required_signatures as f32 / RATE;
129:
130:             let sender_address = tx_message.account_keys[0].to_string();
131:             let receiver_address = tx_message.account_keys[1].to_string();
```

Description

***: The `transaction_handler` function will try to decode the `Transaction`:

```
let decoded_tx_bytes = match tx_encoding {
                TransactionEncoding::Base58 => match bs58::decode(tx_str).into_vec() {
                    Ok(decoded_bytes) => Ok(decoded_bytes),
                    Err(err) => Err(format!("Error decoding base58: {err:?}")),
                },
                TransactionEncoding::Base64 => match base64::decode(tx_str) {
                    Ok(decoded_bytes) => Ok(decoded_bytes),
                    Err(err) => Err(format!("Error decoding base64: {err:?}")),
                },
            };
```

If it fails to decode the `Transaction`, it will use `Default::default()` as the default `Transaction`:

```
let tx: Transaction = match decoded_tx_bytes {
                Ok(decoded_bytes) => {
                    bincode::deserialize(&decoded_bytes).unwrap_or_else(|err| {
                        error!("Error deserializing transaction: {err:?}");
                        // Provide a default value or handle the error case accordingly
                        Default::default()
                    })
                }
                Err(err) => {
                    error!("{err}");
                    // Provide a default value or handle the error case accordingly
                    Default::default()
                }
            };
```

and then it will try to get address from an array:

```
 let tx_message = tx.message();
            let num_required_signatures = tx_message.header.num_required_signatures;

            const BASE_CONGESTION: f32 = 5_000.0;
            const RATE: f32 = 1_000_000_000.0;
            let fee = BASE_CONGESTION * num_required_signatures as f32 / RATE;

            let sender_address = tx_message.account_keys[0].to_string();
            let receiver_address = tx_message.account_keys[1].to_string();
```

The issue here is that if the `Transaction` is a default value, the `account_keys` is an empty array. It will encounter an array out-of-bounds exception in the code `tx_message.account_keys[0]` and the thread will panic.

POC

You can try the following test case:

```
fn main() {
    let tx: Transaction = Default::default();
    let tx_message = tx.message();
    let sender_address = tx_message.account_keys[0].to_string();
    let receiver_address = tx_message.account_keys[1].to_string();
    println!("Hello, world!{:?},{:?}",sender_address,receiver_address);
}
```

This code will encounter an array out-of-bounds exception:

```
thread 'main' panicked at src/main.rs:5:49:
index out of bounds: the len is 0 but the index is 0
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

***: In the transaction_handler function, panic! is used to handle missing parameters and unsupported encoding formats.
panic is not recommended in production code, especially in conversion functions that may handle various external inputs. This disrupts the normal execution process and may result in service interruption.

Recommendation

***: Consider skipping the loop when it fails to decode `Transaction`:

```
let tx: Transaction = match decoded_tx_bytes {
                Ok(decoded_bytes) => {
                    bincode::deserialize(&decoded_bytes).unwrap_or_else(|err| {
                        error!("Error deserializing transaction: {err:?}");
                        // Provide a default value or handle the error case accordingly
                        Default::default()
                    })
                }
                Err(err) => {
                    error!("{err}");
                    // Provide a default value or handle the error case accordingly
                    Default::default()
                }
            };

            if tx == Default::default(){
              continue;
            }
```

***: eplace the panic! calls with appropriate error handling logic that logs the errors and allows the application to continue running.
Code Fix

```
        impl From<&str> for TransactionEncoding {
            fn from(s: &str) -> Self {
                match s {
                    "base58" => Self::Base58,
                    "base64" => Self::Base64,
                    _ => {
                        error!("Unsupported encoding: {}", s);
                        Self::Base58 // Default to Base58 or handle appropriately
                    }
                }
            }
        }
```

Client Response

client response : Fixed. Implemented recommendation of if transaction defaults, skip.
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

client response : Fixed. Removed panic in production code.
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-2:SQL Injection Risk

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/AI-proxy/src/main.rs#L337-L347
- code/AI-proxy/src/main.rs#L451

```
337: // Extract the individual fields from the JSON
338:                 let wallet = json_value["wallet"].as_str().unwrap_or_default();
339:                 let risk_score = json_value["risk_score"].as_f64().unwrap_or_default();
340:                 let ip_address = json_value["ip_address"].as_str().unwrap_or_default();
341:                 let transaction_signature = json_value["transaction_signature"]
342:                     .as_str()
343:                     .unwrap_or_default();
344:                 let timestamp = json_value["timestamp"].as_str().unwrap_or_default();
345:                 let signature = json_value["signature"].as_str().unwrap_or_default();
346:                 let node = json_value["node"].as_str().unwrap_or_default();
347:                 let timeout = json_value["timeout"].as_i64().unwrap_or_default();
```

```
451: async fn update_database(
```

Description

***:

Take a look at https://github.com/Secure3Audit/code_DomiChain_AI/blob/e1498d6b77df311d0da558fcde77e1249fd3134c/code/AI-proxy/src/main.rs#L337-L347

```
                // Extract the individual fields from the JSON
                let wallet = json_value["wallet"].as_str().unwrap_or_default();
                let risk_score = json_value["risk_score"].as_f64().unwrap_or_default();
                let ip_address = json_value["ip_address"].as_str().unwrap_or_default();
                let transaction_signature = json_value["transaction_signature"]
                    .as_str()
                    .unwrap_or_default();
                let timestamp = json_value["timestamp"].as_str().unwrap_or_default();
                let signature = json_value["signature"].as_str().unwrap_or_default();
                let node = json_value["node"].as_str().unwrap_or_default();
                let timeout = json_value["timeout"].as_i64().unwrap_or_default();
```

Evidently, the current code seems to lack proper validation for user input received from requests. This could range to issues in regards to security as malicious actors could potentially inject SQL code or exploit other weaknesses and even issues in regards to Economical exploits/griefing as malicious actors can inject very large data for each field forcing protocol to spend too much when processing their data.

> *"TLDR: Current implementation hints that malicious users could inject SQL code into requests to manipulate database queries and gain unauthorized access to data, alternatively, unsanitized user input can be used for*

XSS attacks, allowing attackers to inject malicious scripts into web pages. And finally, they could inject an extremely large data."

"Potential window for this could also be via this call to `handle_request()` *. other instances include this."*

***: The current implementation of the update_database function in main.rs constructs SQL queries using user-provided input without proper sanitization or escaping. This can lead to SQL injection vulnerabilities, allowing attackers to execute arbitrary SQL commands.

POC :

```
async fn update_database(
    pool: &Pool,
    wallet: &str,
    risk_score: f64,
    ip_address: &str,
    transaction_signature: &str,
    timestamp: &str,
    signature: &str,
    node: &str,
    timeout: i64,
) -> Result<(), MySqlError> {
    let mut conn = pool.get_conn().await?;

    let query = format!(
        "INSERT INTO ip_data (wallet, risk_score, ip_address, transaction_signature, timestamp, signature, nod
e, timeout)
        VALUES ('{}', {}, '{}', '{}', '{}', '{}', '{}', {})",
        wallet, risk_score, ip_address, transaction_signature, timestamp, signature, node, timeout
    );
    conn.query_drop(query).await?;

    Ok(())
}
```

A SQL injection vulnerability allows attackers to manipulate SQL queries by injecting malicious SQL code. This can lead to data breaches, unauthorized access to sensitive information, data corruption, and in severe cases, complete takeover of the database server.

Recommendation

***:

Try to sanitize input data before using it in database queries or other operations. Libraries like `html5escape` can be used to escape special characters in user input (would help for the `XSS` risk). Other implementations could be made to ensure only vouched data are submitted, i.e validate user input based on expected data types and formats. Reject requests with invalid data to prevent unexpected behavior.

***: - Use parameterized queries to ensure that user-provided input is properly sanitized and escaped before being used in SQL queries.

```
async fn update_database(
    pool: &Pool,
    wallet: &str,
    risk_score: f64,
    ip_address: &str,
    transaction_signature: &str,
    timestamp: &str,
    signature: &str,
    node: &str,
    timeout: i64,
) -> Result<(), MySqlError> {
    let mut conn = pool.get_conn().await?;

    let query = r"
        INSERT INTO ip_data (wallet, risk_score, ip_address, transaction_signature, timestamp, signature, node,
timeout)
        VALUES (:wallet, :risk_score, :ip_address, :transaction_signature, :timestamp, :signature, :node, :time
out)
    ";
    conn.exec_drop(
        query,
        params! {
            "wallet" => wallet,
            "risk_score" => risk_score,
            "ip_address" => ip_address,
            "transaction_signature" => transaction_signature,
            "timestamp" => timestamp,
            "signature" => signature,
            "node" => node,
            "timeout" => timeout
        },
    ).await?;

    Ok(())
}
```

By using parameterized queries, we can ensure that user-provided input is properly escaped and sanitized before being used in SQL queries, thereby preventing SQL injection attacks.

Client Response

client response : Fixed. Removal of database and related functions in proxy removes the issue.
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-3:Anyone can potentially bloat the system to hike up the protocol's computational cost due to the way fees are charged

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/AI-proxy/src/ai_handler.rs#L123-L129

```
123: let tx_message = tx.message();
124:             let num_required_signatures = tx_message.header.num_required_signatures;
125:
126:             const BASE_CONGESTION: f32 = 5_000.0;
127:             const RATE: f32 = 1_000_000_000.0;
128:             let fee = BASE_CONGESTION * num_required_signatures as f32 / RATE;
```

Description

***:
Take a look at https://github.com/Secure3Audit/code_DomiChain_AI/blob/e1498d6b77df311d0da558fcde77e1249fd3134c/code/AI-proxy/src/ai_handler.rs#L123-L129

```
            let tx_message = tx.message();
            let num_required_signatures = tx_message.header.num_required_signatures;

            const BASE_CONGESTION: f32 = 5_000.0;
            const RATE: f32 = 1_000_000_000.0;
            let fee = BASE_CONGESTION * num_required_signatures as f32 / RATE;
```

This is how the fee is being computed, which ends up being attached for the specific transaction here, issue with the current implementation however is the fact that there is no base fee, that's to say a griefer can now come and request multiple transactions and passing only 1 signature as the number of their required signature, which then makes the fee to always be equal to:
`5_000 * 2  / 1_000_000_000` = `100_000`.
This is a very low value, which then allows the malicious actor to be able to pass in multiple transactions for as low as this to hike up the protocol's computational cost.

Recommendation

***: Consider having a base fee, i.e in the case there is only one signature attached to `num_required_signatures`, then the fee should be like `1_000_000`, this hikes up the cost to take this attack around `10x`.

Client Response

client response : Fixed. Commit link:3151420f6d05fc5cee5f7edac891475b1336e1de
Base fee implemented:

```
const BASE_CONGESTION: f32 = 5_000.0;
const RATE: f32 = 1_000_000_000.0;
const BASE_FEE: f32 = 1_000_000.0;

                    let fee = BASE_CONGESTION * num_required_signatures as f32 / RATE;
                    let total_fee = if num_required_signatures == 1 {
                        BASE_FEE
                    } else {
                        fee
                    };
```

Notes: This issue isn't a risk as the fees in this step are only used as a reference point for the AI models, actual fee calculation and withdrawal occurs in the validator code.

DMC-4:The function `get_risk_score_by_timestamp` can query datas which `timestamp` are greater than the current time.

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/AI-proxy/src/main.rs#L486-L536

```
async fn get_risk_score_by_timestamp(
    pool: Arc<Pool>,
    time: i64,
) -> MResult<Vec<HashMap<String, String>>> {
    let mut conn = pool.get_conn().await?;
    let query: String = "SELECT wallet, risk_score, transaction_signature, timestamp, signature, node, timeout FROM ip_data".to_string();
    let results_vec: Vec<mysql_async::Row> = conn.query(query).await?;

    let mut results: Vec<HashMap<String, String>> = Vec::new();

    for row in results_vec {
        let wallet: String = row.get("wallet").unwrap();
        let risk_score: f32 = row.get("risk_score").unwrap();
        let transaction_sig: String = row.get("transaction_signature").unwrap();
        let timestamp: String = row.get("timestamp").unwrap();
        let signature: String = row.get("signature").unwrap();
        let node_id: String = row.get("node").unwrap();
        let time_out: i64 = row.get("timeout").unwrap();
        let risk_score_float: f64 = risk_score as f64; // Convert to f64 if it's not already
        let risk_score_str = format!("{:.1}", risk_score_float);
```

```

        let timestamp_as_datetime: DateTime<Utc> = timestamp.parse::<DateTime<Utc>>().unwrap();
        if Utc::now().signed_duration_since(timestamp_as_datetime) < CDuration::seconds(time) {
            let mut entry_map = HashMap::new();
            entry_map.insert("wallet".to_string(), wallet.clone());
            entry_map.insert("risk_score".to_string(), risk_score.to_string());
            entry_map.insert("timeout".to_string(), time_out.to_string());
            entry_map.insert("timestamp".to_string(), timestamp.clone());
            entry_map.insert("signature".to_string(), signature);
            entry_map.insert("node_id".to_string(), node_id);

            let mut data_map = BTreeMap::new();
            data_map.insert("wallet", wallet);
            data_map.insert("risk_score", risk_score_str);
            data_map.insert("transaction_sig", transaction_sig);
            data_map.insert("timestamps", timestamp);
            let data_json = serde_json::to_string(&data_map).unwrap();

            // Turn JSON string into bytes
            let data_bytes = data_json.as_bytes();
```

```
526:
527:             let data_hex = hex::encode(data_bytes);
528:
529:             entry_map.insert("data".to_string(), data_hex);
530:
531:             results.push(entry_map);
532:         }
533:     }
534:
535:     Ok(results)
536: }
```

Description

***: The function `get_risk_score_by_timestamp` querys all the data from mysql and compares it one by one with the input param `time`:

```
let mut conn = pool.get_conn().await?;
    let query: String = "SELECT wallet, risk_score, transaction_signature, timestamp, signature, node, timeo
ut FROM ip_data".to_string();
    let results_vec: Vec<mysql_async::Row> = conn.query(query).await?;

    let mut results: Vec<HashMap<String, String>> = Vec::new();
     for row in results_vec {
        ...
        ...
        let timestamp_as_datetime: DateTime<Utc> = timestamp.parse::<DateTime<Utc>>().unwrap();
        if Utc::now().signed_duration_since(timestamp_as_datetime) < CDuration::seconds(time) {
```

It uses `Utc::now().signed_duration_since(timestamp_as_datetime)` to calculate how long `timestamp_as_datetime` is from the current time and compare it to `time`. The issue here is that If `timestamp_as_datetime` is a future time, then `Utc::now().signed_duration_since(timestamp_as_datetime)` returns a negative number, which is always smaller than `time` and the `if` condition will pass and this data will be included in the returned results.

POC

test case:

```
use chrono::{DateTime, Duration as CDuration, Utc};

fn main() {
    let timestamp_as_datetime = DateTime::parse_from_str("2083 Apr 13 12:09:14.274 +0000", "%Y %b %d %H:%M:%
S%.3f %z").unwrap();
    let s = Utc::now().signed_duration_since(timestamp_as_datetime);
    println!("{:?}",s);
    let pass = s < CDuration::seconds(100);
    println!("{:?}",pass);
}
```

results:

```
TimeDelta { secs: -1857450727, nanos: 811595301 }
true
```

Recommendation

***: Consider filtering out those datas with `timestamp` greater than the current time.

Client Response

client response : Fixed. Removal of database in AI Proxy and Node removes this issue with the database

DMC-5:Missing exception handling

Category	Severity	Client Response	Contributor
DOS	Low	Acknowledged	***

Code Reference

- code/AI-node/Logistic regression.py#L13

```
13: data = pd.read_csv('output.csv')
```

- code/AI-node/main.py#L400-L457

```
400: # Define the "Predict" API endpoint
401: @app.route('/predict', methods=['POST'])
402: async def predict():
403:     data = ip_list_pb2.AIRequest()
404:     data.ParseFromString(request.data)
405:
406:     # Extract transactions from the input
407:     transactions = data.transactions
408:
409:     if len(transactions) <= 20:
410:         for transaction in transactions:
411:             # Extract the IP address from the transaction
412:             ip_address = transaction.requester_ip
413:             value = transaction.transaction_amount
414:             wallet = transaction.sender_address
415:             # Add the IP address to the context list
416:             with context_lock:
417:                 context.append(ip_address)
418:                 ds_context.append([value, wallet])
419:
```

```
420:         if len(context) <= window_size:
421:             return "No predictions made", 200
422:
423:         if len(context) > window_size:
424:
425:             dos_model_predict(context, transactions)
426:             ds_model_predict(ds_context, transactions)
427:
428:             return "Predictions made successfully", 200
429:
430:     elif len(transactions) > 20:
431:         # Add the IP address to the context list
432:         dos_context = []
433:         double_spending_context = []
434:         for transaction in transactions:
435:             # Extract the IP address from the transaction
436:             ip_address = transaction.requester_ip
437:             value = transaction.transaction_amount
438:             wallet = transaction.sender_address
439:
```

```
440:             dos_context.append(ip_address)
441:             double_spending_context.append([value, wallet])
442:
443:         executor.submit(dos_model_predict, dos_context, transactions)
444:         executor.submit(ds_model_predict, double_spending_context, transactions)
445:
446:         return "Prediction made successfully", 200
447:
448:
449: # Define the "Retrieve Risk Score by Timestamp" API endpoint
450: @app.route('/retrieve_risk_score_by_timestamp', methods=['GET'])
451: def retrieve_risk_score_by_timestamp():
452:     time = request.args.get('time')
453:     results = get_risk_score_by_timestamp(time)
454:     if results:
455:         return jsonify(results), 200
456:     else:
457:         return "No recent entries found within the time amount.", 200
```

- code/AI-node/metrics.py#L10

```
10: data = pd.read_csv('output.csv')
```

- code/AI-node/shape.py#L31

```
31: data = pd.read_csv('output.csv')
```

- code/AI-proxy/src/ai_handler.rs#L212-L222

```
212: Err(_) => {
213:                                 // Acquire a lock on shared_ai_node_status_clone
214:                                 let mut ai_node_status = shared_ai_node_status_clone.lock().await;
215:                                 ai_node_status.insert(ai_node_addr_clone, Status::Pending(1));
216:                             }
217:                         }
218:                     });
219:                 }
220:
221:                 ai_trans_count = 0;
222:                 ai_transactions_list.clear();
```

- code/AI-proxy/src/main.rs#L538-L619

```
538: #[tokio::main(flavor = "multi_thread", worker_threads = 8)]
539: async fn main() {
540:     env_logger::init();
541:     let (tx, rx) = mpsc::channel(3000);
542:     let (server_addr, register_addr, domi_node_addr) = parse_cli_arguments();
543:     let ai_node_status: HashMap<String, Status> = HashMap::new();
544:     let shared_ai_node_status = Arc::new(Mutex::new(ai_node_status));
545:     let addr = match server_addr.parse::<SocketAddr>() {
546:         Ok(addr) => addr,
547:         Err(e) => {
548:             error!("Error parsing SocketAddr: {}", e);
549:             return;
550:         }
551:     };
552:
553:     let mut settings = config::Config::default();
554:     settings
555:         .merge(config::File::with_name("config"))
556:         .expect("Error loading config file");
557:
```

```
558:     let settings: Settings = settings.try_into().expect("Error parsing config");
559:     let database_addr = settings.database.addr;
560:     let database_user = settings.database.user;
561:     let database_pass = settings.database.pass;
562:
563:     let opts = OptsBuilder::default()
564:         .db_name(Some("domi"))
565:         .user(Some(database_user))
566:         .pass(Some(database_pass))
567:         .ip_or_hostname(database_addr)
568:         .tcp_port(3306);
569:
570:     let pool = mysql_async::Pool::new(opts);
571:     let pool = Arc::new(pool);
572:
573:     let pool_clone = Arc::clone(&pool);
574:
575:     let register_addr = match register_addr.parse::<SocketAddr>() {
576:         Ok(addr) => addr,
577:         Err(e) => {
```

```
578:             error!("Error parsing SocketAddr: {}", e);
579:             return;
580:         }
581:     };
582:
583:     let client: Arc<Client<HttpConnector, _>> = Arc::new(
584:         Client::builder()
585:             .pool_max_idle_per_host(300)
586:             .pool_idle_timeout(Some(Duration::from_secs(10)))
587:             .build_http(),
588:     );
589:
590:     let tx_handler_client = Arc::clone(&client);
591:     let shared_ai_node_status_clone = Arc::clone(&shared_ai_node_status);
592:     tokio::spawn(async move {
593:         let shared_ai_node_status = shared_ai_node_status_clone;
594:         ai_handler::transaction_handler(tx_handler_client, rx, shared_ai_node_status)
595:             .await
596:             .unwrap();
597:     });
```

```
598:
599:     let shared_ai_node_status_clone = Arc::clone(&shared_ai_node_status);
600:     tokio::spawn(async move {
601:         let shared_ai_node_status = shared_ai_node_status_clone;
602:         run_ai_server(register_addr, shared_ai_node_status, pool).await;
603:     });
604:
605:     let node_health_client = Arc::clone(&client);
606:     let shared_ai_node_status_clone = Arc::clone(&shared_ai_node_status);
607:     tokio::spawn(async move {
608:         let shared_ai_node_status = shared_ai_node_status_clone;
609:         ai_node_manage(node_health_client, shared_ai_node_status).await;
610:     });
611:
612:     // Initialize tables
613:     match create_table(&pool_clone).await {
614:         Ok(_) => println!("Tables created or already exist."),
615:         Err(e) => eprintln!("Error creating table: {:?}", e),
616:     }
617:
```

```
618:     run_server(client, addr, domi_node_addr, tx).await;
619: }
```

Description

***: - The `data.ParseFromString(request.data)` line attempts to parse the request data without any exception handling. If the request data is malformed or invalid, this can raise an exception and cause the application to crash.

- The route does not check if the `time` parameter is provided or if it is in a valid format. This can lead to unexpected behavior or errors when querying the database.

***: The given code snippet attempts to read a CSV file (`output.csv`). If the file does not exist, is inaccessible, or has any issues during reading, the program will raise an exception and terminate unexpectedly.
***: The `main()` function does not handle errors properly when creating tables in the database. If an error occurs during the table creation process, the code simply prints an error message but continues to start the http

server:

```
// Initialize tables
    match create_table(&pool_clone).await {
        Ok(_) => println!("Tables created or already exist."),
        Err(e) => eprintln!("Error creating table: {:?}", e),
    }

    run_server(client, addr, domi_node_addr, tx).await;
```

If the table does not exist, the `update_database` function will always fails to be called.
***:** This code implements sending transactions to each active node. However, when the sending fails, it sets the node status to "pending" and clears all the transactions after executing the process for all nodes. If the sending fails due to network issues in the system itself, it will set all the nodes to "pending" status and clear the transactions, resulting in a direct loss of the transactions. There is no robust error handling mechanism in place.

Recommendation

***:** - Implement try-except blocks to handle potential exceptions during data parsing and query operations.

- Ensure that the request data and query parameters are validated before processing.

```
@app.route('/predict', methods=['POST'])
async def predict():
    data = ip_list_pb2.AIRequest()
    try:
        data.ParseFromString(request.data)
    except Exception as e:
        return jsonify({"error": "Invalid request data", "message": str(e)}), 400
    # ...

@app.route('/retrieve_risk_score_by_timestamp', methods=['GET'])
def retrieve_risk_score_by_timestamp():
    time = request.args.get('time')
    if not time:
        return jsonify({"error": "Missing 'time' parameter"}), 400
    # ...
```

***:**

```
try:
    data = pd.read_csv('output.csv')
except FileNotFoundError:
    print("Error: The file 'output.csv' was not found.")
    exit(1)
```

***:** Consider return when failing to create the table:

```
match create_table(&pool_clone).await {
        Ok(_) => println!("Tables created or already exist."),
        Err(e) => {
          eprintln!("Error creating table: {:?}", e),
          return;
        }
    }
```

***: It is recommended that the code should implement a more robust error handling strategy, such as:

1. Retrying the transaction sending a configurable number of times before marking the node as "pending".
2. Maintaining a queue of failed transactions and periodically attempting to resend them, instead of clearing them completely.
3. Introducing circuit breakers or back-off mechanisms to handle temporary network failures and avoid cascading failures across all nodes.
4. Implementing proper logging and alerting to notify the system owners of transaction failures.

Client Response

client response : Acknowledged.

DMC-6:Lack of Rate Limiting in Proxy Handler

Category	Severity	Client Response	Contributor
DOS	Low	Fixed	***

Code Reference

- code/AI-proxy/src/main.rs#L103

```
103: async fn proxy_handler(
```

Description

***: The current implementation of the `proxy_handler` function in `main.rs` does not implement any rate limiting. This allows an attacker to send a large number of requests in a short period, potentially overwhelming the server and causing a denial of service (DoS).
POC :

```
async fn proxy_handler(
    req: Request<Body>,
    client: Arc<Client<HttpConnector>>,
    remote_addr: SocketAddr,
    domi_node_addr: Arc<String>,
    tx: Sender<AiData>,
) -> Result<Response<Body>, hyper::Error> {
    let headers = req.headers().clone();
    let method = req.method().clone();
    let version = req.version();
    let domi_node_uri = format!("http://{domi_node_addr}");

    let send_request = |body| {
        let mut proxy_req = Request::builder()
            .method(method.clone())
            .uri(domi_node_uri)
            .version(version)
            .body(body)
            .unwrap();

        *proxy_req.headers_mut() = headers.clone();
        proxy_req.headers_mut().insert(HOST, headers[HOST].clone());
```

```
        async {
            let response_result = client.request(proxy_req).await.map_err(|error| {
                eprintln!("Error while making the request: {}", error);
                error
            });
            response_result
        }
    };

    let ((res, _), full_body) = if method == Method::POST {
        let mut full_body = Vec::new();
        let (sender, receiver) = tokio::sync::mpsc::channel(1024);
        let sender = PollSender::new(sender);
        let receiver = ReceiverStream::new(receiver);

        let cloned_sink = SinkExt::fanout(sender.sink_map_err(|e| panic!("{e}")), &mut full_body);
        req.into_body()
            .map(|i| Ok(i.unwrap()))
            .forward(cloned_sink)
            .await
            .unwrap();

        let body = Body::wrap_stream(receiver.map(|i| Ok::<Bytes, &str>(i)));
        (send_request(body).await?, Some(full_body))
    } else {
        (send_request(req.into_body()).await?, None)
    };

    if let Some(full_body) = full_body {
        let full_body: Vec<_> = full_body.into_iter().flatten().collect();
        let send_str = AiData {
            body: full_body,
            remote_ip: remote_addr.ip().to_string(),
        };
        tx.send(send_str).await.unwrap();
    }

    Ok(res)
}
```

A lack of rate limiting can allow an attacker to overwhelm the server with a high volume of requests, causing it to slow down or crash. This can result in legitimate users being unable to access the service, leading to a denial of service (DoS).

Recommendation

***: - Implement rate limiting in the `proxy_handler` function to limit the number of requests an individual client can make in a given period.
Fix :

```
use std::collections::HashMap;
use std::time::{Duration, Instant};
use tokio::sync::Mutex;
use hyper::header::HeaderValue;

struct RateLimiter {
    clients: Mutex<HashMap<SocketAddr, (Instant, u32)>>,
    limit: u32,
    period: Duration,
}

impl RateLimiter {
    fn new(limit: u32, period: Duration) -> Self {
        Self {
            clients: Mutex::new(HashMap::new()),
            limit,
            period,
        }
    }

    async fn is_allowed(&self, addr: &SocketAddr) -> bool {
        let mut clients = self.clients.lock().await;
        let entry = clients.entry(*addr).or_insert((Instant::now(), 0));

        if entry.0.elapsed() > self.period {
            entry.0 = Instant::now();
            entry.1 = 0;
        }

        if entry.1 < self.limit {
            entry.1 += 1;
            true
        } else {
            false
        }
    }
}

let rate_limiter = Arc::new(RateLimiter::new(100, Duration::from_secs(60)));
```

```
async fn proxy_handler(
    req: Request<Body>,
    client: Arc<Client<HttpConnector>>,
    remote_addr: SocketAddr,
    domi_node_addr: Arc<String>,
    tx: Sender<AiData>,
    rate_limiter: Arc<RateLimiter>,
) -> Result<Response<Body>, hyper::Error> {
    if !rate_limiter.is_allowed(&remote_addr).await {
        return Ok(Response::builder()
            .status(429)
            .header("Retry-After", HeaderValue::from_str("60").unwrap())
            .body(Body::from("Rate limit exceeded"))
            .unwrap());
    }

    let headers = req.headers().clone();
    let method = req.method().clone();
    let version = req.version();
    let domi_node_uri = format!("http://{domi_node_addr}");

    let send_request = |body| {
        let mut proxy_req = Request::builder()
            .method(method.clone())
            .uri(domi_node_uri)
            .version(version)
            .body(body)
            .unwrap();

        *proxy_req.headers_mut() = headers.clone();
        proxy_req.headers_mut().insert(HOST, headers[HOST].clone());

        async {
            let response_result = client.request(proxy_req).await.map_err(|error| {
                eprintln!("Error while making the request: {}", error);
                error
            });
            response_result
        }
    };
```

```
    let ((res, _), full_body) = if method == Method::POST {
        let mut full_body = Vec::new();
        let (sender, receiver) = tokio::sync::mpsc::channel(1024);
        let sender = PollSender::new(sender);
        let receiver = ReceiverStream::new(receiver);

        let cloned_sink = SinkExt::fanout(sender.sink_map_err(|e| panic!("{e}")), &mut full_body);
        req.into_body()
            .map(|i| Ok(i.unwrap()))
            .forward(cloned_sink)
            .await
            .unwrap();

        let body = Body::wrap_stream(receiver.map(|i| Ok::<Bytes, &str>(i)));
        (send_request(body).await?, Some(full_body))
    } else {
        (send_request(req.into_body()).await?, None)
    };

    if let Some(full_body) = full_body {
        let full_body: Vec<_> = full_body.into_iter().flatten().collect();
        let send_str = AiData {
            body: full_body,
            remote_ip: remote_addr.ip().to_string(),
        };
        tx.send(send_str).await.unwrap();
    }

    Ok(res)
}
```

By implementing rate limiting, we can prevent denial of service (DoS) attacks by limiting the number of requests an individual client can make in a given period. This helps to ensure that the server remains responsive and available to legitimate users.

Client Response

client response : Fixed. Implemented Rate limiting for proxy and run_server.
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-7:Incorrect http request method

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/AI-node/main.py#L460-L462

```
460: @app.route('/check_health', methods=['GET'])
461: def check_health():
462:     return jsonify({"status": "OK"}), 200
```

- code/AI-proxy/src/ai_node_manager.rs#L65-L80

```
65: pub async fn check_health(
66:     client: Arc<Client<HttpConnector>>,
67:     addr: String,
68: ) -> Result<hyper::Response<Body>, hyper::Error> {
69:     let url = format!("http://{}/check_health", addr);
70:     info!("Send to AI node check_health {:?}", addr);
71:
72:     let req = Request::builder()
73:         .method(Method::POST)
74:         .uri(url)
75:         .header("Content-Type", "application/octet-stream")
76:         .body(Body::from("Check Health"))
77:         .unwrap();
78:
79:     client.request(req).await
80: }
```

Description

***: In `ai_node_manager.rs`, the `check_health` will call the http request with `POST` method:

```
let req = Request::builder()
        .method(Method::POST)
        .uri(url)
        .header("Content-Type", "application/octet-stream")
        .body(Body::from("Check Health"))
        .unwrap();
```

However, in `main.py`, the check health service is defined as a `GET` service:

```
@app.route('/check_health', methods=['GET'])
def check_health():
    return jsonify({"status": "OK"}), 200
```

The request method used in `ai_node_manager.rs` is incorrect.

Recommendation

***: Consider following fix:

```
let req = Request::builder()
        .method(Method::GET)
        .uri(url)
        .header("Content-Type", "application/octet-stream")
        .body(Body::from("Check Health"))
        .unwrap();
```

Client Response

client response : Fixed. Changed check_health to GET from POST as recommended:
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-8:Http response not check

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	***

Code Reference

- code/AI-node/main.py#L457

```
457: return "No recent entries found within the time amount.", 200
```

- code/AI-proxy/src/ai_handler.rs#L205-L217

```
205: match send_to_ai_node(client, data.to_vec(), ai_node_addr).await {
206:                             Ok(_) => {
207:                                 println!(
208:                                     "3....sending {} tx to ai node {:?}",
209:                                     ai_trans_count, ai_node_addr_clone
210:                                 );
211:                             }
212:                             Err(_) => {
213:                                 // Acquire a lock on shared_ai_node_status_clone
214:                                 let mut ai_node_status = shared_ai_node_status_clone.lock().await;
215:                                 ai_node_status.insert(ai_node_addr_clone, Status::Pending(1));
216:                             }
217:                         }
```

- code/AI-proxy/src/ai_node_manager.rs#L36-L53

```
36: match check_health(client, ai_node_addr).await {
37:                     Ok(_) => {
38:                         let mut node_status = shared_ai_node_status.lock().await;
39:                         println!("AI node check health ok {:?}", ai_node_addr_clone);
40:                         node_status.insert(ai_node_addr_clone, Status::Active);
41:                     }
42:                     Err(_) => {
43:                         let mut node_status = shared_ai_node_status.lock().await;
44:                         if let Some(status) = node_status.get_mut(&ai_node_addr_clone) {
45:                             if let Status::Pending(value) = status {
46:                                 *value += 1;
47:                                 if *value >= 5 {
48:                                     node_status.insert(ai_node_addr_clone, Status::Inactive);
49:                                 }
50:                             }
51:                         }
52:                     }
53:                 }
```

- code/AI-proxy/src/main.rs#L300-L311

```
300: if action == "Register" {
301:                     let mut shared_ai_node_status = shared_ai_node_status.lock().await;
302:                     shared_ai_node_status.insert(ai_node_addr, Status::Active);
303:                     println!("AI node registered OK");
304:                 } else {
305:                     return Ok(Response::builder()
306:                         .status(400)
307:                         .body(Body::from("Not found this activity"))
308:                         .unwrap());
309:                 }
310:             }
311:             Ok(Response::new(Body::from("Register Success!")))
```

Description

***: The API endpoint /retrieve_risk_score_by_timestamp returns an HTTP status code of 200 when no recent entries are found within the specified time amount, instead of the correct status code of 404.

```
@app.route('/retrieve_risk_score_by_timestamp', methods=['GET'])
def retrieve_risk_score_by_timestamp(): # @audit-ok
    time = request.args.get('time')
    results = get_risk_score_by_timestamp(time)
    if results:
        return jsonify(results), 200
    else:
        return "No recent entries found within the time amount.", 200
```

When the client receives a 200 status code, it will attempt to decode the response as a JSON object, which will fail because the response is a plain text error message.

```
def retrieve_risk_score_by_timestamp():
    url = "http://localhost:5000/retrieve_risk_score_by_timestamp?time=600"
    response = requests.get(url)

    if response.status_code == 200:
        data = response.json()
        #...
    elif response.status_code == 404:
        print("No recent entries found within the last 10 minutes.")
    else:
        print("Error retrieving risk scores from the server.")
```

***: The `handle_request` function in the `register` endpoint only supports the `POST` method, but it returns a 200 status code when the request method is not `POST`. This inconsistent behavior can lead to confusion and potential security issues. The code snippet below shows the current implementation:

```
match req.uri().path() {
    "/register" => {
        if req.method() == Method::POST {
            //...
        }
        Ok(Response::new(Body::from("Register Success!")))
    }
    //...
}
```

The function returns a "Register Success!" response with a 200 status code even when the request method is not `POST`, which is incorrect.

***: In `ai_node_manager.rs`, the `ai_node_manage` function will call `check_health`:

```
match check_health(client, ai_node_addr).await
```

The `check_health` function will return a result with a http response:

```
client.request(req).await
```

However, the `ai_node_manage` function does not check the response, it ignores the response using `Ok(_)`:

```
match check_health(client, ai_node_addr).await {
                    Ok(_) => {
                        let mut node_status = shared_ai_node_status.lock().await;
                        println!("AI node check health ok {:?}", ai_node_addr_clone);
                        node_status.insert(ai_node_addr_clone, Status::Active);
                    }
                    ...
                }
}
```

If the http code in the response is not 200, this response is not successful and the node should be treated as a failure node.

In `ai_handler.rs`, the same issue exists in the `transaction_handler` function:

```
match send_to_ai_node(client, data.to_vec(), ai_node_addr).await {
                            Ok(_) => {
                                println!(
                                    "3....sending {} tx to ai node {:?}",
                                    ai_trans_count, ai_node_addr_clone
                                );
                            }
                            Err(_) => {
                                // Acquire a lock on shared_ai_node_status_clone
                                let mut ai_node_status = shared_ai_node_status_clone.lock().await;
                                ai_node_status.insert(ai_node_addr_clone, Status::Pending(1));
                            }
                        }
```

Recommendation

***: Return a 404 status code when no recent entries are found within the specified time amount.

```
else:
    return "No recent entries found within the time amount.", 404
```

***: Return a 405 Method Not Allowed status code when the request method is not `POST` for the `register` endpoint. This can be achieved by adding an `else` clause to handle the case where the request method is not `POST`. For example:

```
match req.uri().path() {
    "/register" => {
        if req.method() == Method::POST {
            //...
            if action == "Register" {
                let mut shared_ai_node_status = shared_ai_node_status.lock().await;
                shared_ai_node_status.insert(ai_node_addr, Status::Active);
                println!("AI node registered OK");
                return Ok(Response::new(Body::from("Register Success!")));
            } else {
                return Ok(Response::builder()
                    .status(400)
                    .body(Body::from("Not found this activity"))
                    .unwrap());
            }
        } else {
            Ok(Response::builder()
               .status(405)
               .body(Body::from("Method Not Allowed"))
               .unwrap())
        }
    }
    //...
}
```

***: Consider checking the response in `ai_node_manage` function and `transaction_handler` function, for example:

```
match check_health(client, ai_node_addr).await {
                    Ok(response) => {
                        if !response.status().is_success() {
                            let mut node_status = shared_ai_node_status.lock().await;
                            if let Some(status) = node_status.get_mut(&ai_node_addr_clone) {
                                if let Status::Pending(value) = status {
                                    *value += 1;
                                    if *value >= 5 {
                                        node_status.insert(ai_node_addr_clone, Status::Inactive);
                                    }
                                }
                            }
                        }else{
                            let mut node_status = shared_ai_node_status.lock().await;
                            println!("AI node check health ok {:?}", ai_node_addr_clone);
                            node_status.insert(ai_node_addr_clone, Status::Active);
                        }
                    }
```

```
match send_to_ai_node(client, data.to_vec(), ai_node_addr).await {
                            Ok(response) => {

                                if !response.status().is_success() {
                                    let mut ai_node_status = shared_ai_node_status_clone.lock().await;
                                    ai_node_status.insert(ai_node_addr_clone, Status::Pending(1));
                                }else{
                                    println!(
                                        "3....sending {} tx to ai node {:?}",
                                        ai_trans_count, ai_node_addr_clone
                                    );
                                }
                            }
                            Err(_) => {
                                // Acquire a lock on shared_ai_node_status_clone
                                let mut ai_node_status = shared_ai_node_status_clone.lock().await;
                                ai_node_status.insert(ai_node_addr_clone, Status::Pending(1));
                            }
                        }
```

Client Response

client response : Acknowledged.

DMC-9:Computational transaction with `ai_node_manager#ai_node_manage()` could cause for this function to not work if protocol heaavily scales

Category	Severity	Client Response	Contributor
Logical	Low	Mitigated	***

Code Reference

- code/AI-proxy/src/ai_node_manager.rs#L14-L64

```
14: pub async fn ai_node_manage(
15:     client: Arc<Client<HttpConnector>>,
16:     ai_node_status: Arc<Mutex<HashMap<String, Status>>>,
17: ) {
18:     loop {
19:         let pending_addresses: Vec<String> = ai_node_status
20:             .lock()
21:             .await
22:             .iter()
23:             .filter_map(|(address, status)| match status {
24:                 Status::Pending(value) if (1..=4).contains(value) => Some(address.clone()),
25:                 _ => None,
26:             })
27:             .collect();
28:
29:         for ai_node_addr in pending_addresses {
30:             let client = Arc::clone(&client);
31:             let shared_ai_node_status = Arc::clone(&ai_node_status);
32:
33:             tokio::spawn(async move {
```

```
34:                 let ai_node_addr_clone = ai_node_addr.clone();
35:
36:                 match check_health(client, ai_node_addr).await {
37:                     Ok(_) => {
38:                         let mut node_status = shared_ai_node_status.lock().await;
39:                         println!("AI node check health ok {:?}", ai_node_addr_clone);
40:                         node_status.insert(ai_node_addr_clone, Status::Active);
41:                     }
42:                     Err(_) => {
43:                         let mut node_status = shared_ai_node_status.lock().await;
44:                         if let Some(status) = node_status.get_mut(&ai_node_addr_clone) {
45:                             if let Status::Pending(value) = status {
46:                                 *value += 1;
47:                                 if *value >= 5 {
48:                                     node_status.insert(ai_node_addr_clone, Status::Inactive);
49:                                 }
50:                             }
51:                         }
52:                     }
53:                 }
```

```
54:             });
55:         }
56:
57:         let mut ai_node_status = ai_node_status.lock().await;
58:         ai_node_status.retain(|_, status| *status != Status::Inactive);
59:         drop(ai_node_status);
60:
61:         sleep(Duration::from_secs(60)).await;
62:     }
63: }
```

Description

***:

Take a look at https://github.com/Secure3Audit/code_DomiChain_AI/blob/e1498d6b77df311d0da558fcde77e1249fd3134c/code/AI-proxy/src/ai_node_manager.rs#L14-L64

```rust
pub async fn ai_node_manage(
    client: Arc<Client<HttpConnector>>,
    ai_node_status: Arc<Mutex<HashMap<String, Status>>>,
) {
    loop {
        let pending_addresses: Vec<String> = ai_node_status
            .lock()
            .await
            .iter()
            .filter_map(|(address, status)| match status {
                Status::Pending(value) if (1..=4).contains(value) => Some(address.clone()),
                _ => None,
            })
            .collect();
```

// @audit what's the cost of one attaching multiple pending addresses to make this call to fail... @audit this function lacks any check to ensure the computational cost needed to cover it in one transaction is enough for the amount of elements that need to be looped on, keep in mind thatg the functionality includes a nested loop call to check_Health, which would then mean that having a lot of Ai nodes in the system would make this call revert due to cost, RMS: Consider integrating this functionality in a way that in the case where a lot of ai nodes exist the contract loops through it set by set on different calls where the range is passed.

```
        for ai_node_addr in pending_addresses {
            let client = Arc::clone(&client);
            let shared_ai_node_status = Arc::clone(&ai_node_status);

            tokio::spawn(async move {
                let ai_node_addr_clone = ai_node_addr.clone();

                match check_health(client, ai_node_addr).await {
                    Ok(_) => {
                        let mut node_status = shared_ai_node_status.lock().await;
                        println!("AI node check health ok {:?}", ai_node_addr_clone);
                        node_status.insert(ai_node_addr_clone, Status::Active);
                    }
                    Err(_) => {
                        let mut node_status = shared_ai_node_status.lock().await;
                        if let Some(status) = node_status.get_mut(&ai_node_addr_clone) {
                            if let Status::Pending(value) = status {
                                *value += 1;
                                if *value >= 5 {
                                    node_status.insert(ai_node_addr_clone, Status::Inactive);
                                }
                            }
                        }
                    }
                }
            });
        }

        let mut ai_node_status = ai_node_status.lock().await;
        ai_node_status.retain(|_, status| *status != Status::Inactive);
        drop(ai_node_status);

        sleep(Duration::from_secs(60)).await;
    }
}
```

Short context on what's going on in the for loop:
The code iterates over a list of `pending_addresses` and for each address:

1. Clones references to `client` and `ai_node_status`.
2. Spawns an asynchronous task to call `check_health` for the address.
3. Updates the node's status based on the `check_health` result.

Priblem here is that this function lacks any check to ensure the computational cost needed to cover it in one transaction is enough for the amount of elements that need to be looped on, keep in mind thatg the

functionality includes a nested loop call to `check_Health(`, which would then mean that having a lot of Ai nodes in the system would make this call revert due to cost, RMS:

Recommendation

***:** Consider integrating this functionality in a way that in the case where a lot of ai nodes exist the contract loops through it set by set on different calls where the range is passed.
By processing addresses in smaller batches, the gas cost per transaction can be controlled. The system can also handle a larger number of AI nodes without encountering transaction cost limitations. And most importantly, batching reduces the risk of transaction failures due to exceeding the gas limit.

Client Response

client response : Mitigated. These api calls and checks do not run on the blockchain itself and as such gas fees are not an issue. However potentially batching could speed up this process.

DMC-10:When there is too much data in the database, the http request may timeout

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/AI-node/main.py#L71-L128

```
71: def get_risk_score_by_timestamp(time):
72:
73:     try:
74:         with getConnection() as conn:
75:             cursor = conn.cursor()
76:             query = """
77:                 SELECT wallet, risk_score, transaction_signature, timestamp, signature, node, timeout
78:                 FROM ip_data
79:             """
80:             cursor.execute(query)
81:             rows = cursor.fetchall()
82:
83:             results = []
84:
85:             for row in rows:
86:                 wallet = row[0]
87:                 risk_score = row[1]
88:                 transaction_sig = row[2]
89:                 timestamps = row[3]
90:                 signature = row[4]
```

```
91:                 node_id = row[5]
92:                 time_out = row[6]
93:
94:                 data_dict = {
95:                     "wallet": wallet,
96:                     "risk_score": str(risk_score),
97:                     "transaction_sig": transaction_sig,
98:                     "timestamps": timestamps
99:                 }
100:
101:                 data_bytes = json.dumps(data_dict).encode()
102:                 data = data_bytes.hex()
103:
104:                 # Check if there are timestamps and if the latest timestamp is within the time range
105:                 if datetime.fromisoformat(timestamps) >= \
106:                         (datetime.now(timezone.utc) - timedelta(seconds=int(time))):
107:
108:                     select_query = "SELECT public_key FROM node_public_keys WHERE node_id = %s"
109:                     cursor.execute(select_query, (node_id,))
110:                     row = cursor.fetchone()
```

```
111:                     public_key = row[0]
112:
113:                     # Append the result as a dictionary to the list
114:                     result = {
115:                         'risk_score': risk_score,
116:                         'wallet': wallet,
117:                         'data': data,
118:                         'timeout': time_out,
119:                         'timestamp': timestamps,
120:                         'public_key': public_key,
121:                         'signature': signature
122:                     }
123:                     results.append(result)
124:
125:             return results
126:     except Error as e:
127:         print("Error retrieving data from the database:", str(e))
128:         return []
```

- code/AI-proxy/src/main.rs#L486-L536

```
486: async fn get_risk_score_by_timestamp(
487:     pool: Arc<Pool>,
488:     time: i64,
489: ) -> MResult<Vec<HashMap<String, String>>> {
490:     let mut conn = pool.get_conn().await?;
491:     let query: String = "SELECT wallet, risk_score, transaction_signature, timestamp, signature, node, ti
meout FROM ip_data".to_string();
492:     let results_vec: Vec<mysql_async::Row> = conn.query(query).await?;
493:
494:     let mut results: Vec<HashMap<String, String>> = Vec::new();
495:
496:     for row in results_vec {
497:         let wallet: String = row.get("wallet").unwrap();
498:         let risk_score: f32 = row.get("risk_score").unwrap();
499:         let transaction_sig: String = row.get("transaction_signature").unwrap();
500:         let timestamp: String = row.get("timestamp").unwrap();
501:         let signature: String = row.get("signature").unwrap();
502:         let node_id: String = row.get("node").unwrap();
503:         let time_out: i64 = row.get("timeout").unwrap();
504:         let risk_score_float: f64 = risk_score as f64; // Convert to f64 if it's not already
505:         let risk_score_str = format!("{:.1}", risk_score_float);
```

```
506:
507:         let timestamp_as_datetime: DateTime<Utc> = timestamp.parse::<DateTime<Utc>>().unwrap();
508:         if Utc::now().signed_duration_since(timestamp_as_datetime) < CDuration::seconds(time) {
509:             let mut entry_map = HashMap::new();
510:             entry_map.insert("wallet".to_string(), wallet.clone());
511:             entry_map.insert("risk_score".to_string(), risk_score.to_string());
512:             entry_map.insert("timeout".to_string(), time_out.to_string());
513:             entry_map.insert("timestamp".to_string(), timestamp.clone());
514:             entry_map.insert("signature".to_string(), signature);
515:             entry_map.insert("node_id".to_string(), node_id);
516:
517:             let mut data_map = BTreeMap::new();
518:             data_map.insert("wallet", wallet);
519:             data_map.insert("risk_score", risk_score_str);
520:             data_map.insert("transaction_sig", transaction_sig);
521:             data_map.insert("timestamps", timestamp);
522:             let data_json = serde_json::to_string(&data_map).unwrap();
523:
524:             // Turn JSON string into bytes
525:             let data_bytes = data_json.as_bytes();
```

```
526:
527:             let data_hex = hex::encode(data_bytes);
528:
529:             entry_map.insert("data".to_string(), data_hex);
530:
531:             results.push(entry_map);
532:         }
533:     }
534:
535:     Ok(results)
536: }
```

Description

***: The function `get_risk_score_by_timestamp` querys all the data from mysql and compares it one by one with the input param `time`:

```
let mut conn = pool.get_conn().await?;
    let query: String = "SELECT wallet, risk_score, transaction_signature, timestamp, signature, node, timeo
ut FROM ip_data".to_string();
    let results_vec: Vec<mysql_async::Row> = conn.query(query).await?;

    let mut results: Vec<HashMap<String, String>> = Vec::new();
     for row in results_vec {
        ...
        ...
        let timestamp_as_datetime: DateTime<Utc> = timestamp.parse::<DateTime<Utc>>().unwrap();
        if Utc::now().signed_duration_since(timestamp_as_datetime) < CDuration::seconds(time) {
```

The issue here is that if there are too much data in the database, the query time will be big and all client requests are likely to be timeout.

Recommendation

***: Consider using time comparisons in sql.

Client Response

client response : Acknowledged.

DMC-11:Using `unwrap()` could lead to the application to panic

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/ai_handler.rs#L76-L245

```
76: let request: JsonRpcRequest = serde_json::from_slice(&request_str.body).unwrap();
77:         dbg!(&request);
78:
79:         // Start prepare the data for ai
80:         if request.method == "sendTransaction" {
81:             let params_array = request.params.unwrap();
82:             let tx_str = params_array
83:                 .get(0)
84:                 .unwrap_or_else(|| panic!("No transaction in params"))
85:                 .as_str()
86:                 .unwrap();
87:
88:             let tx_options = params_array.get(1);
89:             let tx_encoding = match tx_options {
90:                 Some(options) => match options.get("encoding") {
91:                     Some(encoding) => TransactionEncoding::from(encoding.as_str().unwrap()),
92:                     None => TransactionEncoding::Base58,
93:                 },
94:                 None => TransactionEncoding::Base58,
95:             };
```

```
96:
97:             let decoded_tx_bytes = match tx_encoding {
98:                 TransactionEncoding::Base58 => match bs58::decode(tx_str).into_vec() {
99:                     Ok(decoded_bytes) => Ok(decoded_bytes),
100:                     Err(err) => Err(format!("Error decoding base58: {err:?}")),
101:                 },
102:                 TransactionEncoding::Base64 => match base64::decode(tx_str) {
103:                     Ok(decoded_bytes) => Ok(decoded_bytes),
104:                     Err(err) => Err(format!("Error decoding base64: {err:?}")),
105:                 },
106:             };
107:
108:             let tx: Transaction = match decoded_tx_bytes {
109:                 Ok(decoded_bytes) => {
110:                     bincode::deserialize(&decoded_bytes).unwrap_or_else(|err| {
111:                         error!("Error deserializing transaction: {err:?}");
112:                         // Provide a default value or handle the error case accordingly
113:                         Default::default()
114:                     })
115:                 }
```

```
116:                 Err(err) => {
117:                     error!("{err}");
118:                     // Provide a default value or handle the error case accordingly
119:                     Default::default()
120:                 }
121:             };
122:
123:             let tx_message = tx.message();
124:             let num_required_signatures = tx_message.header.num_required_signatures;
125:
126:             const BASE_CONGESTION: f32 = 5_000.0;
127:             const RATE: f32 = 1_000_000_000.0;
128:             let fee = BASE_CONGESTION * num_required_signatures as f32 / RATE;
129:
130:             let sender_address = tx_message.account_keys[0].to_string();
131:             let receiver_address = tx_message.account_keys[1].to_string();
132:
133:             let (transaction_amount, instructions) =
134:                 if let Some(instruction) = tx_message.instructions.first() {
135:                     let data = instruction.data.as_slice();
```

```
136:                     let lamports = u64::from_le_bytes(data[4..12].try_into().unwrap());
137:                     //println!("Transaction Amount {:?}", lamports as f32 / RATE);
138:                     //println!("instructions {:?}", *instruction);
139:                     (lamports as f32 / RATE, format!("{instruction:?}"))
140:                 } else {
141:                     Default::default()
142:                 };
143:             //println!("requester_ip {:?}", remote_addr);
144:             let hash = tx.signatures[0].to_string();
145:             //let request_ip = remote_addr.ip().to_string();
146:             let requester_ip = request_str.remote_ip;
147:
148:             /* let mut ai_transaction: AITransaction = AITransaction::default();
149:             ai_transaction.hash = hash.to_string();
150:             ai_transaction.fee = fee;
151:             ai_transaction.transaction_amount = transaction_amount;
152:             ai_transaction.sender_address = sender_address;
153:             ai_transaction.receiver_address = receiver_address;
154:             ai_transaction.instructions = instruction_string;
155:             ai_transaction.requester_ip = request_ip; */
```

```
156:
157:             let ai_transaction = AITransaction {
158:                 hash,
159:                 fee,
160:                 transaction_amount,
161:                 sender_address,
162:                 receiver_address,
163:                 instructions,
164:                 requester_ip,
165:                 special_fields: Default::default(),
166:             };
167:
168:             ai_trans_count += 1;
169:             ai_transactions_list.push(ai_transaction);
170:
171:             let ai_node_status = shared_ai_node_status.lock().await;
172:
173:             // Collect active addresses
174:             let active_addresses: Vec<String> = ai_node_status
175:                 .iter()
```

```
176:                 .filter(|(_, status)| **status == Status::Active)
177:                 .map(|(address, _)| address.clone())
178:                 .collect();
179:
180:             // Release the lock to avoid holding it while performing async operations
181:             drop(ai_node_status);
182:
183:             // Check if ai_trans_count is 30
184:             if ai_trans_count >= 30 {
185:                 println!("1...sending {} tx to ai node", ai_trans_count);
186:
187:                 let ai_transactions_list_clone = ai_transactions_list.clone();
188:                 let mut ai_request = AIRequest::default();
189:                 ai_request.transactions = ai_transactions_list_clone;
190:                 let serialized_ai_request = ai_request
191:                     .write_to_bytes()
192:                     .expect("Failed to serialize message");
193:                 let shared_serialized_ai_request = Arc::new(serialized_ai_request);
194:                 for ai_node_addr in active_addresses {
195:                     let client = Arc::clone(&client);
```

```
196:                     let data = shared_serialized_ai_request.clone();
197:                     let shared_ai_node_status_clone = Arc::clone(&shared_ai_node_status);
198:
199:                     tokio::spawn(async move {
200:                         let ai_node_addr_clone = ai_node_addr.clone();
201:                         println!(
202:                             "2...sending {} tx to ai node {:?}",
203:                             ai_trans_count, ai_node_addr_clone
204:                         );
205:                         match send_to_ai_node(client, data.to_vec(), ai_node_addr).await {
206:                             Ok(_) => {
207:                                 println!(
208:                                     "3....sending {} tx to ai node {:?}",
209:                                     ai_trans_count, ai_node_addr_clone
210:                                 );
211:                             }
212:                             Err(_) => {
213:                                 // Acquire a lock on shared_ai_node_status_clone
214:                                 let mut ai_node_status = shared_ai_node_status_clone.lock().await;
215:                                 ai_node_status.insert(ai_node_addr_clone, Status::Pending(1));
```

```
216:                             }
217:                         }
218:                     });
219:                 }
220:
221:                 ai_trans_count = 0;
222:                 ai_transactions_list.clear();
223:             }
224:         }
225:     }
226: }
227:
228: pub async fn send_to_ai_node(
229:     client: Arc<Client<HttpConnector>>,
230:     serialized_ai_request: Vec<u8>,
231:     ai_node_addr: String,
232: ) -> Result<hyper::Response<Body>, hyper::Error> {
233:     let url = format!("{ai_node_addr}/predict");
234:     info!("Send to AI node add_context {url:?}");
235:
```

```
236:     let req = Request::builder()
237:         .method(Method::POST)
238:         .uri(url)
239:         .header("Content-Type", "application/octet-stream")
240:         .body(Body::from(serialized_ai_request))
241:         .unwrap();
242:
243:     client.request(req).await
244: }
```

- code/AI-proxy/src/main.rs#L97-L596
- code/AI-proxy/src/main.rs#L117-L127

```
97: matches.value_of("proxy_addr").unwrap().to_string(),
98:         matches.value_of("register_addr").unwrap().to_string(),
99:         matches.value_of("domi_node_addr").unwrap().to_string(),
100:     )
101: }
102:
103: async fn proxy_handler(
104:     req: Request<Body>,
105:     client: Arc<Client<HttpConnector>>,
106:     remote_addr: SocketAddr,
107:     domi_node_addr: Arc<String>,
108:     tx: Sender<AiData>,
109: ) -> Result<Response<Body>, hyper::Error> {
110:     //let request_start = Instant::now();
111:     let headers = req.headers().clone();
112:     let method = req.method().clone();
113:     let version = req.version();
114:
115:     let domi_node_uri = format!("http://{domi_node_addr}");
116:
```

```
117:     let send_request = |body| {
118:         let mut proxy_req = Request::builder()
119:             .method(method.clone())
120:             .uri(domi_node_uri)
121:             .version(version)
122:             .body(body)
123:             .unwrap();
124:
125:         *proxy_req.headers_mut() = headers.clone();
126:         proxy_req.headers_mut().insert(HOST, headers[HOST].clone());
127:
128:         async {
129:             let sending_request = Instant::now();
130:             let response_result = client.request(proxy_req).await.map_err(|error| {
131:                 eprintln!("Error while making the request: {}", error);
132:                 error
133:             });
134:             let sending_request_time = dbg!(sending_request.elapsed());
135:             response_result.map(|resp| (resp, sending_request_time))
136:         }
```

```
137:     };
138:
139:     let ((res, _), full_body) = if method == Method::POST {
140:         let mut full_body = Vec::new();
141:
142:         // let (sender, receiver) = broadcast::channel(1024);
143:
144:         let (sender, receiver) = tokio::sync::mpsc::channel(1024);
145:         let sender = PollSender::new(sender);
146:         let receiver = ReceiverStream::new(receiver);
147:
148:         let cloned_sink = SinkExt::fanout(sender.sink_map_err(|e| panic!("{e}")), &mut full_body);
149:
150:         // FIXME join!{} with send_request future
151:         req.into_body()
152:             .map(|i| Ok(i.unwrap()))
153:             .forward(cloned_sink)
154:             .await
155:             .unwrap();
156:
```

```
157:         let body = Body::wrap_stream(receiver.map(|i| Ok::<Bytes, &str>(i)));
158:
159:         (send_request(body).await?, Some(full_body))
160:     } else {
161:         (send_request(req.into_body()).await?, None)
162:     };
163:
164:     if let Some(full_body) = full_body {
165:         dbg!(&full_body);
166:         let full_body: Vec<_> = full_body.into_iter().flatten().collect();
167:
168:         let constructing_send_string = Instant::now();
169:         let send_str = AiData {
170:             body: full_body,
171:             remote_ip: remote_addr.ip().to_string(),
172:         };
173:         tokio::spawn(async move {
174:             let sending_to_channel = Instant::now();
175:             if let Err(e) = tx.send(send_str).await {
176:                 eprintln!("Error sending data through channel: {}", e);
```

```
177:             }
178:             dbg!(sending_to_channel.elapsed());
179:             dbg!(tx.capacity());
180:         });
181:         dbg!(constructing_send_string.elapsed());
182:     }
183:     Ok(res)
184: }
185:
186: async fn run_server(
187:     client: Arc<Client<HttpConnector>>,
188:     addr: SocketAddr,
189:     domi_node_addr: String,
190:     tx: Sender<AiData>,
191: ) {
192:     let domi_node_addr = Arc::new(domi_node_addr);
193:
194:     // Create a closure to build the Hyper service
195:     let make_svc = make_service_fn(|conn: &AddrStream| {
196:         let client = Arc::clone(&client);
```

```
197:         let remote_addr = conn.remote_addr();
198:         let domi_node_addr = Arc::clone(&domi_node_addr);
199:         let tx = tx.clone(); // Clone the sender for use in the closure
200:
201:         async move {
202:             let domi_node_addr = Arc::clone(&domi_node_addr);
203:
204:             Ok::<_, Infallible>(service_fn(move |req| {
205:                 proxy_handler(
206:                     req,
207:                     Arc::clone(&client),
208:                     remote_addr,
209:                     Arc::clone(&domi_node_addr),
210:                     tx.clone(),
211:                 )
212:             }))
213:         }
214:     });
215:
216:     // Create the Hyper server and bind it to the specified address
```

```
217:     let server = Server::bind(&addr).serve(make_svc);
218:
219:     info!("Proxy server listening on http://{}", addr);
220:
221:     // Start the server and await for it to finish
222:     if let Err(e) = server.await {
223:         error!("Server error: {}", e);
224:     }
225: }
226:
227: async fn run_ai_server(
228:     addr: SocketAddr,
229:     shared_ai_node_status: Arc<Mutex<HashMap<String, Status>>>,
230:     pool: Arc<Pool>,
231: ) {
232:     let shared_ai_node_status = Arc::clone(&shared_ai_node_status);
233:     // Create a closure to build the Hyper service
234:     let make_svc = make_service_fn(|_| {
235:         let pool = pool.clone();
236:         let shared_ai_node_status = Arc::clone(&shared_ai_node_status);
```

```
237:         async move {
238:             let svc_result = Ok::<_, Infallible>(service_fn(move |req| {
239:                 handle_request(req, shared_ai_node_status.clone(), pool.clone())
240:             }));
241:             svc_result
242:         }
243:     });
244:
245:     // Create the Hyper server and bind it to the specified address
246:     let server = Server::bind(&addr).serve(make_svc);
247:
248:     info!("Proxy server listening on http://{addr}");
249:
250:     // Start the server and await for it to finish
251:     if let Err(e) = server.await {
252:         error!("Server error: {e}");
253:     }
254: }
255:
256: async fn handle_request(
```

```
    req: Request<Body>,
    shared_ai_node_status: Arc<Mutex<HashMap<String, Status>>>,
    pool: Arc<Pool>,
) -> Result<Response<Body>, Infallible> {
    // Handle your requests here based on the req.uri() path
    match req.uri().path() {
        "/register" => {
            if req.method() == Method::POST {
                let json_str = match hyper::body::to_bytes(req.into_body()).await {
                    Ok(bytes) => bytes,
                    Err(err) => {
                        error!("Error reading request body: {}", err);
                        return Ok(Response::builder()
                            .status(500)
                            .body(Body::from("Internal Server Error"))
                            .unwrap());
                    }
                };

                // struct
```

```
                // struct Ai {
                //     activity: String,
                //     url: String,
                // }

                let json_value: Value = match serde_json::from_slice(&json_str) {
                    Ok(value) => value,
                    Err(err) => {
                        error!("Error parsing JSON: {}", err);
                        return Ok(Response::builder()
                            .status(400) // Bad Request
                            .body(Body::from("Invalid JSON"))
                            .unwrap());
                    }
                };

                println!("json body is {:?}", json_value[0]);

                let action = json_value[0]["activity"].as_str().unwrap().to_string();
                let ai_node_addr = json_value[0]["url"].as_str().unwrap().to_string();
```

```
297:                 println!("activity is {:?}", action);
298:                 println!("url is {:?}", ai_node_addr);
299:
300:                 if action == "Register" {
301:                     let mut shared_ai_node_status = shared_ai_node_status.lock().await;
302:                     shared_ai_node_status.insert(ai_node_addr, Status::Active);
303:                     println!("AI node registered OK");
304:                 } else {
305:                     return Ok(Response::builder()
306:                         .status(400)
307:                         .body(Body::from("Not found this activity"))
308:                         .unwrap());
309:                 }
310:             }
311:             Ok(Response::new(Body::from("Register Success!")))
312:         }
313:         "/update" => {
314:             if req.method() == Method::POST {
315:                 let json_str = match hyper::body::to_bytes(req.into_body()).await {
316:                     Ok(bytes) => bytes,
```

```
317:                     Err(err) => {
318:                         error!("Error reading request body: {}", err);
319:                         return Ok(Response::builder()
320:                             .status(500)
321:                             .body(Body::from("Internal Server Error"))
322:                             .unwrap());
323:                     }
324:                 };
325:
326:                 let json_value: Value = match serde_json::from_slice(&json_str) {
327:                     Ok(value) => value,
328:                     Err(err) => {
329:                         error!("Error parsing JSON: {}", err);
330:                         return Ok(Response::builder()
331:                             .status(400)
332:                             .body(Body::from("Invalid JSON"))
333:                             .unwrap());
334:                     }
335:                 };
336:
```

```
337:                 // Extract the individual fields from the JSON
338:                 let wallet = json_value["wallet"].as_str().unwrap_or_default();
339:                 let risk_score = json_value["risk_score"].as_f64().unwrap_or_default();
340:                 let ip_address = json_value["ip_address"].as_str().unwrap_or_default();
341:                 let transaction_signature = json_value["transaction_signature"]
342:                     .as_str()
343:                     .unwrap_or_default();
344:                 let timestamp = json_value["timestamp"].as_str().unwrap_or_default();
345:                 let signature = json_value["signature"].as_str().unwrap_or_default();
346:                 let node = json_value["node"].as_str().unwrap_or_default();
347:                 let timeout = json_value["timeout"].as_i64().unwrap_or_default();
348:
349:                 // Call the database update function
350:                 let pool = Arc::clone(&pool);
351:                 match update_database(
352:                     &pool,
353:                     wallet,
354:                     risk_score,
355:                     ip_address,
356:                     transaction_signature,
```

```
357:                     timestamp,
358:                     signature,
359:                     node,
360:                     timeout,
361:                 )
362:                 .await
363:                 {
364:                     Ok(_) => println!(""),
365:                     Err(e) => eprintln!("Error inserting into database: {:?}", e),
366:                 }
367:
368:                 Ok(Response::new(Body::from("Update Success!")))
369:             } else {
370:                 Ok(Response::builder()
371:                     .status(405)
372:                     .body(Body::from("Method Not Allowed"))
373:                     .unwrap())
374:             }
375:         }
376:         "/retrieve_risk_score_by_timestamp" => {
```

```
377:             if req.method() == Method::GET {
378:                 println!("Received time parameter: {:?}", req.uri().query());
379:                 let mut time: i64 = 60; // Default value
380:                 if let Some(query) = req.uri().query() {
381:                     let pairs: Vec<&str> = query.split('&').collect();
382:                     for pair in pairs {
383:                         let key_value: Vec<&str> = pair.split('=').collect();
384:                         if key_value.len() == 2 {
385:                             if key_value[0] == "time" {
386:                                 if let Ok(parsed_time) = key_value[1].parse::<i64>() {
387:                                     time = parsed_time;
388:                                 } else {
389:                                     return Ok(Response::builder()
390:                                         .status(400)
391:                                         .body(Body::from("Bad Request: Invalid time parameter"))
392:                                         .unwrap());
393:                                 }
394:                             }
395:                         }
396:                     }
```

```
397:                 }
398:                 let results = get_risk_score_by_timestamp(Arc::clone(&pool), time).await;
399:
400:                 match results {
401:                     Ok(data) => {
402:                         let json_data = serde_json::to_string(&data).unwrap();
403:                         Ok(Response::builder()
404:                             .status(200)
405:                             .body(Body::from(json_data))
406:                             .unwrap())
407:                     }
408:                     Err(_) => Ok(Response::builder()
409:                         .status(500)
410:                         .body(Body::from("Internal Server Error"))
411:                         .unwrap()),
412:                 }
413:             } else {
414:                 Ok(Response::builder()
415:                     .status(405)
416:                     .body(Body::from("Method Not Allowed"))
```

```
                    .unwrap())
            }
        }
        _ => {
            // Return a 404 response for any other path
            Ok(Response::builder()
                .status(404)
                .body(Body::from("Not Found"))
                .unwrap())
        }
    }
}

async fn create_table(pool: &Pool) -> Result<(), Box<dyn std::error::Error>> {
    let mut conn = pool.get_conn().await?;

    // Create the ip_data table
    conn.query_drop(
        r"CREATE TABLE IF NOT EXISTS ip_data (
            wallet VARCHAR(255),
```

```
            risk_score FLOAT,
            ip_address VARCHAR(255),
            transaction_signature VARCHAR(255),
            timestamp VARCHAR(255),
            signature VARCHAR(255),
            node VARCHAR(255),
            timeout VARCHAR(255)
        )",
    )
    .await?;

    Ok(())
}

async fn update_database(
    pool: &Pool,
    wallet: &str,
    risk_score: f64,
    ip_address: &str,
    transaction_signature: &str,
```

```
457:     timestamp: &str,
458:     signature: &str,
459:     node: &str,
460:     timeout: i64,
461: ) -> Result<(), MySqlError> {
462:     let mut conn = pool.get_conn().await?;
463:
464:     let query = r"
465:         INSERT INTO ip_data (wallet, risk_score, ip_address, transaction_signature, timestamp, signature,
node, timeout)
466:         VALUES (:wallet, :risk_score, :ip_address, :transaction_signature, :timestamp, :signature, :node,
:timeout)
467:     ";
468:     conn.exec_drop(
469:         query,
470:         params! {
471:             "wallet" => wallet,
472:             "risk_score" => risk_score,
473:             "ip_address" => ip_address,
474:             "transaction_signature" => transaction_signature,
475:             "timestamp" => timestamp,
476:             "signature" => signature,
```

```
477:             "node" => node,
478:             "timeout" => timeout
479:         },
480:     )
481:     .await?;
482:
483:     Ok(())
484: }
485:
486: async fn get_risk_score_by_timestamp(
487:     pool: Arc<Pool>,
488:     time: i64,
489: ) -> MResult<Vec<HashMap<String, String>>> {
490:     let mut conn = pool.get_conn().await?;
491:     let query: String = "SELECT wallet, risk_score, transaction_signature, timestamp, signature, node, ti
meout FROM ip_data".to_string();
492:     let results_vec: Vec<mysql_async::Row> = conn.query(query).await?;
493:
494:     let mut results: Vec<HashMap<String, String>> = Vec::new();
495:
496:     for row in results_vec {
```

```
497:         let wallet: String = row.get("wallet").unwrap();
498:         let risk_score: f32 = row.get("risk_score").unwrap();
499:         let transaction_sig: String = row.get("transaction_signature").unwrap();
500:         let timestamp: String = row.get("timestamp").unwrap();
501:         let signature: String = row.get("signature").unwrap();
502:         let node_id: String = row.get("node").unwrap();
503:         let time_out: i64 = row.get("timeout").unwrap();
504:         let risk_score_float: f64 = risk_score as f64; // Convert to f64 if it's not already
505:         let risk_score_str = format!("{:.1}", risk_score_float);
506:
507:         let timestamp_as_datetime: DateTime<Utc> = timestamp.parse::<DateTime<Utc>>().unwrap();
508:         if Utc::now().signed_duration_since(timestamp_as_datetime) < CDuration::seconds(time) {
509:             let mut entry_map = HashMap::new();
510:             entry_map.insert("wallet".to_string(), wallet.clone());
511:             entry_map.insert("risk_score".to_string(), risk_score.to_string());
512:             entry_map.insert("timeout".to_string(), time_out.to_string());
513:             entry_map.insert("timestamp".to_string(), timestamp.clone());
514:             entry_map.insert("signature".to_string(), signature);
515:             entry_map.insert("node_id".to_string(), node_id);
516:
```

```
517:             let mut data_map = BTreeMap::new();
518:             data_map.insert("wallet", wallet);
519:             data_map.insert("risk_score", risk_score_str);
520:             data_map.insert("transaction_sig", transaction_sig);
521:             data_map.insert("timestamps", timestamp);
522:             let data_json = serde_json::to_string(&data_map).unwrap();
523:
524:             // Turn JSON string into bytes
525:             let data_bytes = data_json.as_bytes();
526:
527:             let data_hex = hex::encode(data_bytes);
528:
529:             entry_map.insert("data".to_string(), data_hex);
530:
531:             results.push(entry_map);
532:         }
533:     }
534:
535:     Ok(results)
536: }
```

```
537:
538: #[tokio::main(flavor = "multi_thread", worker_threads = 8)]
539: async fn main() {
540:     env_logger::init();
541:     let (tx, rx) = mpsc::channel(3000);
542:     let (server_addr, register_addr, domi_node_addr) = parse_cli_arguments();
543:     let ai_node_status: HashMap<String, Status> = HashMap::new();
544:     let shared_ai_node_status = Arc::new(Mutex::new(ai_node_status));
545:     let addr = match server_addr.parse::<SocketAddr>() {
546:         Ok(addr) => addr,
547:         Err(e) => {
548:             error!("Error parsing SocketAddr: {}", e);
549:             return;
550:         }
551:     };
552:
553:     let mut settings = config::Config::default();
554:     settings
555:         .merge(config::File::with_name("config"))
556:         .expect("Error loading config file");
```

```
557:
558:     let settings: Settings = settings.try_into().expect("Error parsing config");
559:     let database_addr = settings.database.addr;
560:     let database_user = settings.database.user;
561:     let database_pass = settings.database.pass;
562:
563:     let opts = OptsBuilder::default()
564:         .db_name(Some("domi"))
565:         .user(Some(database_user))
566:         .pass(Some(database_pass))
567:         .ip_or_hostname(database_addr)
568:         .tcp_port(3306);
569:
570:     let pool = mysql_async::Pool::new(opts);
571:     let pool = Arc::new(pool);
572:
573:     let pool_clone = Arc::clone(&pool);
574:
575:     let register_addr = match register_addr.parse::<SocketAddr>() {
576:         Ok(addr) => addr,
```

```
577:         Err(e) => {
578:             error!("Error parsing SocketAddr: {}", e);
579:             return;
580:         }
581:     };
582:
583:     let client: Arc<Client<HttpConnector, _>> = Arc::new(
584:         Client::builder()
585:             .pool_max_idle_per_host(300)
586:             .pool_idle_timeout(Some(Duration::from_secs(10)))
587:             .build_http(),
588:     );
589:
590:     let tx_handler_client = Arc::clone(&client);
591:     let shared_ai_node_status_clone = Arc::clone(&shared_ai_node_status);
592:     tokio::spawn(async move {
593:         let shared_ai_node_status = shared_ai_node_status_clone;
594:         ai_handler::transaction_handler(tx_handler_client, rx, shared_ai_node_status)
595:             .await
596:             .unwrap();
```

```
117: let send_request = |body| {
118:         let mut proxy_req = Request::builder()
119:             .method(method.clone())
120:             .uri(domi_node_uri)
121:             .version(version)
122:             .body(body)
123:             .unwrap();
124:
125:         *proxy_req.headers_mut() = headers.clone();
126:         proxy_req.headers_mut().insert(HOST, headers[HOST].clone());
```

Description

***:

Take a look at https://github.com/Secure3Audit/code_DomiChain_AI/blob/e1498d6b77df311d0da558fcde77e1249fd3134c/code/AI-proxy/src/main.rs#L117-L127

```
    let send_request = |body| {
        let mut proxy_req = Request::builder()
            .method(method.clone())
            .uri(domi_node_uri)
            .version(version)
            .body(body)
            .unwrap();

        *proxy_req.headers_mut() = headers.clone();
        proxy_req.headers_mut().insert(HOST, headers[HOST].clone());
```

Here we have the `unwrap` on the data to be processed so as to construct a builder object to configure a request, however using unwrap is unsafe.

In Rust, some functions could return an `Result<T, E>` type, indicating either a successful outcome with the value of type `T` (the request builder in this case) or an error of type `E`.

Now `unwrap()` attempts to extract the value from the `Result`. If it's a `Ok(value)`, it unwraps the value and assigns it to the variable `proxy_req`. However, if it's an `Err(error)`, `unwrap` panics, causing the program to crash with an error message.
Problem with this implementation is that If `Request::builder` encounters any errors during request creation (e.g., invalid URI), `unwrap` will cause a panic which halts the application and doesn't even provide much information about the specific error.

***: `serde_json::from_slice(&request_str.body).unwrap();` This line of code will cause the program to crash if deserialization fails.

Recommendation

***:
Consider replacing unwrap with an implementation that entails better error handling so the application doesn't panic.
***: The ? operator or explicit error handling to handle possible errors.
Similarly, other unwrap operations should be replaced with safer error handling.

Client Response

client response : Fixed. Added proper error handling for deserialization.
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-12:Use of requests without the timeout argument

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/AI-node/main.py#L469

```
469: response = requests.post(register_url, json=data)
```

Description

***: The Python requests library is used in the AI-node without the timeout argument. By default, the requests library will wait until the connection is closed before fulfilling a request. Without the timeout argument, the program will hang indefinitely.

```
response = requests.post(register_url, json=data)
```

Recommendation

***: Add the timeout argument to code locations indicated above. This will ensure that the code will not hang if the website being requested hangs.

Client Response

client response : Acknowledged.

DMC-13:The use of `.get()` to read array data can lead to out-of-bounds access

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/ai_handler.rs#L82-L89

```
82: let tx_str = params_array
83:                 .get(0)
84:                 .unwrap_or_else(|| panic!("No transaction in params"))
85:                 .as_str()
86:                 .unwrap();
87:
88:             let tx_options = params_array.get(1);
89:             let tx_encoding = match tx_options {
```

Description

***: **Out-of-bounds array access** occurs when a program attempts to read or write to memory locations outside the bounds of an allocated array.
In Rust, out-of-bounds array access can lead to memory corruption, data leakage, and potentially exploitable security vulnerabilities. These vulnerabilities typically arise from improper bounds checking or unchecked array accesses.
`transaction_handler()` function attempts to access an element of the array at an index 0 and 1. The code fails to validate if a value actually exits.If not, This can lead to undefined behavior, memory corruption, or potential security vulnerabilities if sensitive data is accessed or overwritten.
For more information, Read the blog - https://blog.devsecopsguides.com/attacking-rust#heading-out-of-bounds-array-access

Recommendation

***: Please find a sample fix of the code wherin the `get(index)`` is gracefully handled
The Fix

```
fn compliant_out_of_bounds_access() {
    let array = [1, 2, 3, 4, 5];
    let index = 2; // Accessing within bounds
    if let Some(value) = array.get(index) {
        println!("Value at index {}: {}", index, value);
    } else {
        println!("Index {} is out of bounds", index);
    }
}
```

Client Response

client response : Fixed. Fixed and aimed to more gracefully handle out of bounds
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-14:The Rust contracts are using outdated crates with known vulnerabilities

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/AI-proxy/Cargo.lock#L1

```
1: # This file is automatically @generated by Cargo.
```

Description

***: The contracts under `AI-Proxy` use external crates or libraries as part of the project. Some of these crates are outdated and can be easily exploited. The attack types include** *Denial of Service (DoS), segmentation faults, resource exhaustion**, etc. In total, 12 vulnerabilities have been found.
Below are some of the affected RUST advisories:

```
https://rustsec.org/advisories/RUSTSEC-2024-0336
https://rustsec.org/advisories/RUSTSEC-2024-0332
https://rustsec.org/advisories/RUSTSEC-2024-0320
https://rustsec.org/advisories/RUSTSEC-2024-0019
https://rustsec.org/advisories/RUSTSEC-2024-0006
https://rustsec.org/advisories/RUSTSEC-2024-0003
https://rustsec.org/advisories/RUSTSEC-2023-0072
```

Proof of Concept:
Run `cargo audit` in the directory where the `Cargo.lock` file is present to reproduce the issue.

Recommendation

***: A quick fix would be to update these crates to the latest version. Running the command `cargo audit fix` should do it, but it's better to run a dry run before updating all the packages as some dependency issues might pop up. Read the full guide here for further assistance: Introducing cargo-audit-fix and more.

Client Response

client response : Acknowledged.

DMC-15:Sensitive Data Exposure

Category	Severity	Client Response	Contributor
Write to Arbitrary Storage Location	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/main.rs#L164

```
164: if let Some(full_body) = full_body {
```

Description

***: The current implementation logs the full request body, which can contain sensitive information. This sensitive data exposure can lead to significant security issues if the logs are accessed by unauthorized users.
POC :

```
if let Some(full_body) = full_body {
    dbg!(&full_body);
    let full_body: Vec<_> = full_body.into_iter().flatten().collect();
}
```

Impact:

Logging sensitive data can lead to unauthorized access to sensitive information, such as transaction details, user information, or authentication tokens. This can result in significant security breaches and privacy violations.

Recommendation

***: - Avoid logging sensitive data. If logging is necessary, ensure that sensitive data is masked or removed before logging. Implement proper access controls to log files to prevent unauthorized access.

```
if let Some(full_body) = full_body {
    // Avoid logging sensitive data
    // Remove or mask sensitive information if necessary
    let full_body: Vec<_> = full_body.into_iter().flatten().collect();

    // Example of masking sensitive information
    let masked_body: Vec<_> = full_body.iter().map(|chunk| {
        if chunk_contains_sensitive_data(chunk) {
            mask_sensitive_data(chunk)
        } else {
            chunk.clone()
        }
    }).collect();

    // If absolutely necessary to log, ensure it's done securely
    // dbg!(&masked_body); // Remove or securely handle this line
}

// Helper functions for masking sensitive data
fn chunk_contains_sensitive_data(chunk: &Bytes) -> bool {
    // Implement logic to detect sensitive data
    // For example, regex matching for patterns like keys or passwords
    false
}

fn mask_sensitive_data(chunk: &Bytes) -> Bytes {
    // Implement logic to mask or remove sensitive data
    // For example, replacing sensitive parts with asterisks
    Bytes::from("****")
}
```

By avoiding the logging of sensitive data, we reduce the risk of exposing sensitive information to unauthorized users. If logging is necessary, sensitive data should be masked or removed to protect user privacy and maintain security.

Client Response

client response : Fixed. Removed Debug print statements as logging is not needed in production
Commit: https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-16:Return value is never checked

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/AI-node/main.py#L51

```
51: pool_name="ip_database_pool",
```

- code/AI-node/request_prediction.py#L51

```
51: response = requests.post(predict_url, data=predictions_data.SerializeToString(), headers={'Content-Type':
'application/octet-stream'})
```

Description

***: In the file `main.py`, the function `register_node` does a post request but never checks if the response is valid or invalid.
For example, if the response's http status code is 400 or 500, which stands for a client error or a server error, the program should throw the error or add other business to cover the request failue case.
Similar case exists in the `request_prediction.py` file, line 51.

Recommendation

***: Consider checking the request response's status (code), throwing error if the request fails or adding other business to cover the request failue case.

Client Response

client response : Acknowledged.

DMC-17:Protocol currently hardciodes the mspc channel capacity

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/AI-proxy/src/main.rs#L539-L551

```
539: async fn main() {
540:     env_logger::init();
541:     let (tx, rx) = mpsc::channel(3000);
542:     let (server_addr, register_addr, domi_node_addr) = parse_cli_arguments();
543:     let ai_node_status: HashMap<String, Status> = HashMap::new();
544:     let shared_ai_node_status = Arc::new(Mutex::new(ai_node_status));
545:     let addr = match server_addr.parse::<SocketAddr>() {
546:         Ok(addr) => addr,
547:         Err(e) => {
548:             error!("Error parsing SocketAddr: {}", e);
549:             return;
550:         }
551:     };
```

Description

***:

Take a look at https://github.com/Secure3Audit/code_DomiChain_AI/blob/e1498d6b77df311d0da558fcde77e1249fd3134c/code/AI-proxy/src/main.rs#L539-L551

```
async fn main() {
    env_logger::init();
    let (tx, rx) = mpsc::channel(3000);//@audit
    let (server_addr, register_addr, domi_node_addr) = parse_cli_arguments();
    let ai_node_status: HashMap<String, Status> = HashMap::new();
    let shared_ai_node_status = Arc::new(Mutex::new(ai_node_status));
    let addr = match server_addr.parse::<SocketAddr>() {
        Ok(addr) => addr,
        Err(e) => {
            error!("Error parsing SocketAddr: {}", e);
            return;
        }
    };
```

Evidently, the current implementation uses an mpsc channel with a fixed capacity of `3000` for communication between the proxy and the transaction handler service. While this might seem sufficient for protocol initially, it's crucial to evaluate its adequacy under varying traffic loads and when protocol has plans to heavily scale in more users. That's to say, currently if the channel reaches its capacity with older data and new data arrives, messages might be dropped, leading to data loss. Also when protocol has heavily scaled up this might lead to the channel being easily fille, which causes the proxy to block waiting to send new data, impacting overall performance.

Recommendation

***:
Explore alternative channel implementations that allow for dynamic capacity adjustments.

Client Response

client response : Acknowledged.

DMC-18:Predictable Random Values: Potential Duplicate Values in Repeated Runs

Category	Severity	Client Response	Contributor
Weak Sources of Randomness	Informational	Acknowledged	***

Code Reference

- code/AI-node/request_prediction.py#L18

```
18: def generate_random_ip():
```

Description

***: The current implementation of random value generation for transaction data could potentially generate duplicate values on repeated runs, which may not accurately simulate unique transactions.
POC :

```
import random

def generate_random_ip():
    return f'{random.randint(1, 255)}.{random.randint(1, 255)}.{random.randint(1, 255)}.{random.randint(1, 255)}'
```

This can lead to unrealistic test data and potentially skew the evaluation of the system's performance.

Recommendation

***: Use a combination of random values and UUIDs to ensure uniqueness across repeated runs.
import random
import uuid

Improved implementation

```
def generate_random_ip():
    return f'{random.randint(1, 255)}.{random.randint(1, 255)}.{random.randint(1, 255)}.{random.randint(1, 255)}-{uuid.uuid4()}'
```

By appending a UUID to the generated IP address, we ensure each IP address is unique even across repeated runs.

Client Response

client response : Acknowledged. This issue is valid, but is not needed to be fixed as this is mainly a test script for a system check: ensuring AI node/models predict functions are working as intended, therefore unique values are not required and duplicates are not a concern.

DMC-19:Potential Race Condition in `ds_model_predict` and `dos_model_predict`

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/AI-node/main.py#L273-L275
- code/AI-node/main.py#L334-L336

```
273: X_processed = []
274:     for i in range(window_size, len(X_test)):
275:         X_processed.append(X_test[i - window_size:i])
```

```
334: X_sequences = []
335:     for i in range(window_size, X_processed.shape[0]):
336:         X_sequences.append(X_processed[i - window_size:i])
```

Description

***:

The function `ds_model_predict` leverages a global variable `ds_context` to calculate the risk score.

```
ds_context = []
context_lock = Lock()
```

The `ds_model_predict` function uses the context_lock to ensure that the length of ds_context is equal to or smaller than window_size when popping elements.

```
    while len(ds_context) > window_size:
        with context_lock:
            ds_context.pop(0)
```

However, it does not use the lock when accessing x, which is the passed value from ds_context. This can lead to a race condition where the length of X_test is modified concurrently, causing the for loop that appends elements to X_processed to iterate over an empty range.

```
@app.route('/predict', methods=['POST'])
async def predict():
    #...
        if len(context) > window_size:

            dos_model_predict(context, transactions)
            ds_model_predict(ds_context, transactions)

            return "Predictions made successfully", 200

def ds_model_predict(x, transaction_list):
    encoder = LabelEncoder()
    wallets = []
    values = []

    # Label encode the columns with strings
    for item in x:
        value, wallet = item
        wallets.append(encoder.fit_transform([str(wallet)]))
        values.append([value])
    X_test = []
    for item in zip(wallets, values):
        X_test.append(item)

    X_processed = []
    for i in range(window_size, len(X_test)): # @audit missing check if window_size > len(X_test)
        X_processed.append(X_test[i - window_size:i])

    X_processed = np.array(X_processed)

    # Extract values from X_processed
    X_values = X_processed[:, :, 0]  # Extract the first column (values)
    X_values = X_values.reshape(-1, window_size, 1)
```

An empty array is not allowed in NumPy. An error like: `IndexError: too many indices for array: array is 0-dimensional, but 3 were indexed` will be raised, which causes a failure of the function `ds_model_predict`.
The same issue exists in the function `dos_model_predict`.

Recommendation

***: Use the context_lock when accessing x to ensure thread safety.
Check if window_size is greater than the length of X_test before iterating over it to prevent the IndexError.
Example:

```
with context_lock:
    for item in x:
        #...

if window_size > len(X_test):
    # Handle the error or return a default value
else:
    for i in range(window_size, len(X_test)):
        X_processed.append(X_test[i - window_size:i])
```

Client Response

client response : Acknowledged.

DMC-20:Potential Denial of Service (DoS) Attack in AI-proxy/src/main.rs

Category	Severity	Client Response	Contributor
DOS	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/main.rs#L139-L162

```
139: let ((res, _), full_body) = if method == Method::POST {
140:         let mut full_body = Vec::new();
141:
142:         // let (sender, receiver) = broadcast::channel(1024);
143:
144:         let (sender, receiver) = tokio::sync::mpsc::channel(1024);
145:         let sender = PollSender::new(sender);
146:         let receiver = ReceiverStream::new(receiver);
147:
148:         let cloned_sink = SinkExt::fanout(sender.sink_map_err(|e| panic!("{e}")), &mut full_body);
149:
150:         // FIXME join!{} with send_request future
151:         req.into_body()
152:             .map(|i| Ok(i.unwrap()))
153:             .forward(cloned_sink)
154:             .await
155:             .unwrap();
156:
157:         let body = Body::wrap_stream(receiver.map(|i| Ok::<Bytes, &str>(i)));
158:
```

```
159:         (send_request(body).await?, Some(full_body))
160:     } else {
161:         (send_request(req.into_body()).await?, None)
162:     };
```

Description

***: The proxy_handler function in the AI Proxy Server project currently collects the entire request body into a Vec using the following code:

```
let ((res, _), full_body) = if method == Method::POST {
    let mut full_body = Vec::new();
    // let (sender, receiver) = broadcast::channel(1024);
    let (sender, receiver) = tokio::sync::mpsc::channel(1024);
    let sender = PollSender::new(sender);
    let receiver = ReceiverStream::new(receiver);
    let cloned_sink = SinkExt::fanout(sender.sink_map_err(|e| panic!("{e}")), &mut full_body);
    req.into_body().map(|i| Ok(i.unwrap())).forward(cloned_sink).await.unwrap();
    let body = Body::wrap_stream(receiver.map(|i| Ok::<Bytes, &str>(i)));
    (send_request(body).await?, Some(full_body))
} else {
    (send_request(req.into_body()).await?, None)
};
```

This approach can lead to memory exhaustion if the request body is very large, potentially resulting in a Denial of Service (DoS) attack.
Detailed Analysis
By collecting the entire request body into a Vec, the function is vulnerable to scenarios where an attacker sends a very large request body, causing the server to allocate large amounts of memory. This can exhaust the server's memory resources, leading to degraded performance or even crashes.
Potential Impact
Memory Exhaustion: Collecting large request bodies into memory can exhaust the available memory, causing the server to crash or become unresponsive.
Denial of Service: An attacker could exploit this vulnerability to perform a DoS attack, disrupting service availability.

Recommendation

***: To mitigate the risk of DoS attacks, it's crucial to handle request bodies in a streaming fashion rather than loading them entirely into memory. This can be achieved by using hyper's Body to stream data directly.

Client Response

client response : Fixed. Changed to streaming however some loading into memory is required for sending data to AI nodes, mitigated so only valid POST methods will be loaded to memory while all else is streamed.
Commit:
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-21:Optimizing Resource Usage When Processing Fewer Transactions

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/ai_handler.rs#L171-L223

```
171: let ai_node_status = shared_ai_node_status.lock().await;
172:
173:             // Collect active addresses
174:             let active_addresses: Vec<String> = ai_node_status
175:                 .iter()
176:                 .filter(|(_, status)| **status == Status::Active)
177:                 .map(|(address, _)| address.clone())
178:                 .collect();
179:
180:             // Release the lock to avoid holding it while performing async operations
181:             drop(ai_node_status);
182:
183:             // Check if ai_trans_count is 30
184:             if ai_trans_count >= 30 {
185:                 println!("1...sending {} tx to ai node", ai_trans_count);
186:
187:                 let ai_transactions_list_clone = ai_transactions_list.clone();
188:                 let mut ai_request = AIRequest::default();
189:                 ai_request.transactions = ai_transactions_list_clone;
190:                 let serialized_ai_request = ai_request
```

```
191:                     .write_to_bytes()
192:                     .expect("Failed to serialize message");
193:                 let shared_serialized_ai_request = Arc::new(serialized_ai_request);
194:                 for ai_node_addr in active_addresses {
195:                     let client = Arc::clone(&client);
196:                     let data = shared_serialized_ai_request.clone();
197:                     let shared_ai_node_status_clone = Arc::clone(&shared_ai_node_status);
198:
199:                     tokio::spawn(async move {
200:                         let ai_node_addr_clone = ai_node_addr.clone();
201:                         println!(
202:                             "2...sending {} tx to ai node {:?}",
203:                             ai_trans_count, ai_node_addr_clone
204:                         );
205:                         match send_to_ai_node(client, data.to_vec(), ai_node_addr).await {
206:                             Ok(_) => {
207:                                 println!(
208:                                     "3....sending {} tx to ai node {:?}",
209:                                     ai_trans_count, ai_node_addr_clone
210:                                 );
```

```
211:                             }
212:                             Err(_) => {
213:                                 // Acquire a lock on shared_ai_node_status_clone
214:                                 let mut ai_node_status = shared_ai_node_status_clone.lock().await;
215:                                 ai_node_status.insert(ai_node_addr_clone, Status::Pending(1));
216:                             }
217:                         }
218:                     });
219:                 }
220:
221:                 ai_trans_count = 0;
222:                 ai_transactions_list.clear();
223:             }
```

Description

***: This code first acquires a shared lock on the ai_node_status, then finds the active nodes. It then checks the number of transactions, and if it's greater than 30, it sends them to the active nodes for processing. However, when there are less than 30 transactions, it needs to acquire the shared lock, find the active nodes, and then release the lock each time. This wastes some system resources. In reality, when there are less than 30 transactions, there is no need to find the active nodes.

Recommendation

***: The logic of acquiring the lock, finding the active nodes, and then releasing the lock should be placed after the logic of if ai_trans_count >= 30. This way, when the condition to send the transactions is met, the system will then acquire the lock, get the active nodes, and then send the transactions.

```
if ai_trans_count >= 30 {

  let ai_node_status = shared_ai_node_status.lock().await;

  let active_addresses: Vec<String> = ai_node_status.iter().filter(|(_, status)| **status == Status::Activ
e).map(|(address, _)| address.clone()).collect();

  drop(ai_node_status);

  // ...
}
```

Client Response

client response : Fixed. Fixed and implemented recommendation, in latest commit: https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-22:Missing `Pool::disconnect` in `main` Function

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/main.rs#L570-L573

```
570: let pool = mysql_async::Pool::new(opts);
571:     let pool = Arc::new(pool);
572:
573:     let pool_clone = Arc::clone(&pool);
```

Description

***:

The `main` function creates a connection pool to connect to a MySQL database using `mysql_async::Pool::new(opts)`. The pool is then cloned and used in two spawned tasks. However, the `main` function does not await `Pool::disconnect` before dropping the runtime, which may result in a number of connections that are not cleanly terminated. This can lead to resource leaks and potential security issues.

```
let pool = mysql_async::Pool::new(opts);
let pool = Arc::new(pool);
//...
// No await on Pool::disconnect before dropping the runtime
```

Recommendation

***: Recommended to await `Pool::disconnect` before dropping the runtime. This can be achieved by calling `pool.disconnect().await` before the end of the `main` function. This will ensure that all connections are cleanly terminated, preventing resource leaks and potential security issues.
See https://docs.rs/mysql_async/latest/mysql_async/#example

Client Response

client response : Fixed. MySQL and related database functions have been removed in the latest commit.
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-23:Missing Event Emissions can lead to No Transparency and Traceability in case of Security Incident

Category	Severity	Client Response	Contributor
Language Specific	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/main.rs#L451-L483

```
451: async fn update_database(
452:     pool: &Pool,
453:     wallet: &str,
454:     risk_score: f64,
455:     ip_address: &str,
456:     transaction_signature: &str,
457:     timestamp: &str,
458:     signature: &str,
459:     node: &str,
460:     timeout: i64,
461: ) -> Result<(), MySqlError> {
462:     let mut conn = pool.get_conn().await?;
463:
464:     let query = r"
465:         INSERT INTO ip_data (wallet, risk_score, ip_address, transaction_signature, timestamp, signature,
node, timeout)
466:         VALUES (:wallet, :risk_score, :ip_address, :transaction_signature, :timestamp, :signature, :node,
:timeout)
467:     ";
468:     conn.exec_drop(
469:         query,
470:         params! {
```

```
471:             "wallet" => wallet,
472:             "risk_score" => risk_score,
473:             "ip_address" => ip_address,
474:             "transaction_signature" => transaction_signature,
475:             "timestamp" => timestamp,
476:             "signature" => signature,
477:             "node" => node,
478:             "timeout" => timeout
479:         },
480:     )
481:     .await?;
482:
483:     Ok(())
```

Description

***: In the context of an L1 blockchain processing multiple requests, the absence of emitting **events** in contracts can significantly hinder **transparency** and **traceability**. Events serve as a crucial mechanism for logging significant occurrences and state changes within the blockchain. Without them, it becomes challenging for external observers, including dApp users, developers, and auditors, to monitor contract activities, verify transaction outcomes, or debug issues efficiently.

This can reduce trust in the system, complicate troubleshooting, and make it harder to identify and respond to anomalies or malicious activities in real-time.

Recommendation

***: To understand event emission, you can include a logging mechanism. Below a code sample from the existing code from the contract

```rust
use log::{info, error};

async fn update_database( //@audit-ok -> the functions are private by default
    pool: &Pool,
    wallet: &str,
    risk_score: f64,
    ip_address: &str,
    transaction_signature: &str,
    timestamp: &str,
    signature: &str,
    node: &str,
    timeout: i64,
) -> Result<(), MySqlError> {
    let mut conn = pool.get_conn().await?;

    let query = r"
        INSERT INTO ip_data (wallet, risk_score, ip_address, transaction_signature, timestamp, signature, node, timeout)
        VALUES (:wallet, :risk_score, :ip_address, :transaction_signature, :timestamp, :signature, :node, :timeout)
    ";

    match conn.exec_drop(
        query,
        params! {
            "wallet" => wallet,
            "risk_score" => risk_score,
            "ip_address" => ip_address,
            "transaction_signature" => transaction_signature,
            "timestamp" => timestamp,
            "signature" => signature,
            "node" => node,
            "timeout" => timeout
        },
    )
    .await {
        Ok(_) => {
            info!("Database update successful for wallet: {}, ip_address: {}", wallet, ip_address);
            Ok(())
        },
        Err(e) => {
            error!("Database update failed for wallet: {}, ip_address: {}. Error: {:?}", wallet, ip_address, e);
            Err(e)
        }
    }
}
```

This code uses the `log` crate to emit informational and error events. In a more advanced scenario, you might replace these logs with blockchain-specific event emission mechanisms to ensure that the events are recorded on-chain and can be queried by users and other contracts.

Client Response

client response : Fixed. Removal of the database and now using transactions in the following commit will have data recorded on chain.
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-24:Lack of authentication for http server

Category	Severity	Client Response	Contributor
DOS	Informational	Acknowledged	***

Code Reference

- code/AI-node/main.py#L400-L478

```
400: # Define the "Predict" API endpoint
401: @app.route('/predict', methods=['POST'])
402: async def predict():
403:     data = ip_list_pb2.AIRequest()
404:     data.ParseFromString(request.data)
405:
406:     # Extract transactions from the input
407:     transactions = data.transactions
408:
409:     if len(transactions) <= 20:
410:         for transaction in transactions:
411:             # Extract the IP address from the transaction
412:             ip_address = transaction.requester_ip
413:             value = transaction.transaction_amount
414:             wallet = transaction.sender_address
415:             # Add the IP address to the context list
416:             with context_lock:
417:                 context.append(ip_address)
418:                 ds_context.append([value, wallet])
419:
```

```
420:         if len(context) <= window_size:
421:             return "No predictions made", 200
422:
423:         if len(context) > window_size:
424:
425:             dos_model_predict(context, transactions)
426:             ds_model_predict(ds_context, transactions)
427:
428:             return "Predictions made successfully", 200
429:
430:     elif len(transactions) > 20:
431:         # Add the IP address to the context list
432:         dos_context = []
433:         double_spending_context = []
434:         for transaction in transactions:
435:             # Extract the IP address from the transaction
436:             ip_address = transaction.requester_ip
437:             value = transaction.transaction_amount
438:             wallet = transaction.sender_address
439:
```

```
            dos_context.append(ip_address)
            double_spending_context.append([value, wallet])

        executor.submit(dos_model_predict, dos_context, transactions)
        executor.submit(ds_model_predict, double_spending_context, transactions)

        return "Prediction made successfully", 200


# Define the "Retrieve Risk Score by Timestamp" API endpoint
@app.route('/retrieve_risk_score_by_timestamp', methods=['GET'])
def retrieve_risk_score_by_timestamp():
    time = request.args.get('time')
    results = get_risk_score_by_timestamp(time)
    if results:
        return jsonify(results), 200
    else:
        return "No recent entries found within the time amount.", 200


```

```
@app.route('/check_health', methods=['GET'])
def check_health():
    return jsonify({"status": "OK"}), 200


def register_node():
    url = f"http://{connection_host}:{connection_port}"
    data = [{'activity': 'Register', 'url': url}]
    try:
        response = requests.post(register_url, json=data)
    except requests.exceptions.RequestException as e:
        print("Error registering node:", str(e))


if __name__ == '__main__':
    insert_public_key_from_env()
    register_node()
    # app.run(debug=True)
    serve(app, host=connection_host, port=connection_port, threads=32)
```

- code/AI-proxy/src/main.rs#L186-L254

```
186: async fn run_server(
187:     client: Arc<Client<HttpConnector>>,
188:     addr: SocketAddr,
189:     domi_node_addr: String,
190:     tx: Sender<AiData>,
191: ) {
192:     let domi_node_addr = Arc::new(domi_node_addr);
193:
194:     // Create a closure to build the Hyper service
195:     let make_svc = make_service_fn(|conn: &AddrStream| {
196:         let client = Arc::clone(&client);
197:         let remote_addr = conn.remote_addr();
198:         let domi_node_addr = Arc::clone(&domi_node_addr);
199:         let tx = tx.clone(); // Clone the sender for use in the closure
200:
201:         async move {
202:             let domi_node_addr = Arc::clone(&domi_node_addr);
203:
204:             Ok::<_, Infallible>(service_fn(move |req| {
205:                 proxy_handler(
206:                     req,
207:                     Arc::clone(&client),
208:                     remote_addr,
209:                     Arc::clone(&domi_node_addr),
210:                     tx.clone(),
211:                 )
212:             }))
213:         }
214:     });
215:
216:     // Create the Hyper server and bind it to the specified address
217:     let server = Server::bind(&addr).serve(make_svc);
218:
219:     info!("Proxy server listening on http://{}", addr);
220:
221:     // Start the server and await for it to finish
222:     if let Err(e) = server.await {
223:         error!("Server error: {}", e);
224:     }
225: }
```

```
226:
227: async fn run_ai_server(
228:     addr: SocketAddr,
229:     shared_ai_node_status: Arc<Mutex<HashMap<String, Status>>>,
230:     pool: Arc<Pool>,
231: ) {
232:     let shared_ai_node_status = Arc::clone(&shared_ai_node_status);
233:     // Create a closure to build the Hyper service
234:     let make_svc = make_service_fn(|_| {
235:         let pool = pool.clone();
236:         let shared_ai_node_status = Arc::clone(&shared_ai_node_status);
237:         async move {
238:             let svc_result = Ok::<_, Infallible>(service_fn(move |req| {
239:                 handle_request(req, shared_ai_node_status.clone(), pool.clone())
240:             }));
241:             svc_result
242:         }
243:     });
244:
245:     // Create the Hyper server and bind it to the specified address
```

```
246:     let server = Server::bind(&addr).serve(make_svc);
247:
248:     info!("Proxy server listening on http://{addr}");
249:
250:     // Start the server and await for it to finish
251:     if let Err(e) = server.await {
252:         error!("Server error: {e}");
253:     }
254: }
```

Description

***: In `main.py`, the `__main__` function will start a server to handle http request:

```
if __name__ == '__main__':
    insert_public_key_from_env()
    register_node()
    # app.run(debug=True)
    serve(app, host=connection_host, port=connection_port, threads=32)
```

The issue is that it lacks of authentication mechanism. If a hacker discovers the IP and port of this server, he can directly invoke the http request to perform dangerous operations, for example, executing the `/predict` request to execute prediction on AI model. An attacker can keep calling this API, which in turn causes the service to run out of resources and not be able to provide normal service.
The same issue exists in `run_server` and `run_ai_server` functions in `main.rs`.

Recommendation

***: Consider using JWT or making sure the api can only be called by the trusted users.

Client Response

client response : Acknowledged.

DMC-25:It is recommended to add a request limiter to the `predict` function

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/AI-node/main.py#L400-L446

```
400: # Define the "Predict" API endpoint
401: @app.route('/predict', methods=['POST'])
402: async def predict():
403:     data = ip_list_pb2.AIRequest()
404:     data.ParseFromString(request.data)
405:
406:     # Extract transactions from the input
407:     transactions = data.transactions
408:
409:     if len(transactions) <= 20:
410:         for transaction in transactions:
411:             # Extract the IP address from the transaction
412:             ip_address = transaction.requester_ip
413:             value = transaction.transaction_amount
414:             wallet = transaction.sender_address
415:             # Add the IP address to the context list
416:             with context_lock:
417:                 context.append(ip_address)
418:                 ds_context.append([value, wallet])
419:
```

```
420:         if len(context) <= window_size:
421:             return "No predictions made", 200
422:
423:         if len(context) > window_size:
424:
425:             dos_model_predict(context, transactions)
426:             ds_model_predict(ds_context, transactions)
427:
428:             return "Predictions made successfully", 200
429:
430:     elif len(transactions) > 20:
431:         # Add the IP address to the context list
432:         dos_context = []
433:         double_spending_context = []
434:         for transaction in transactions:
435:             # Extract the IP address from the transaction
436:             ip_address = transaction.requester_ip
437:             value = transaction.transaction_amount
438:             wallet = transaction.sender_address
439:
```

```
440:             dos_context.append(ip_address)
441:             double_spending_context.append([value, wallet])
442:
443:         executor.submit(dos_model_predict, dos_context, transactions)
444:         executor.submit(ds_model_predict, double_spending_context, transactions)
445:
446:         return "Prediction made successfully", 200
```

Description

***: It is recommended to add a request limiter to the `predict` function to prevent abuse or excessive requests. Without a request limiter, an attacker could potentially flood the endpoint with a large number of requests, causing resource exhaustion or denial of service.

Recommendation

***: To add a request limiter, you can use a library like Flask-Limiter or Flask-Limiter-Cache. These libraries provide rate limiting functionality and allow you to set limits on the number of requests per minute, hour, or day.

Client Response

client response : Acknowledged.

DMC-26:Insecure Storage of Database Credentials

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/AI-node/main.py#L31

```
31: db_password = config.get('Database', 'password')
```

- code/AI-proxy/src/main.rs#L561

```
561: let database_pass = settings.database.pass;
```

Description

***: The database password is stored in plain text in the `config.ini` file, which is a security risk. The `configparser` module is used to read the configuration file, and the database credentials are retrieved without any encryption or protection. This means that an attacker with access to the file system can easily obtain the database password.

```
db_password = config.get('Database', 'password')
```

see https://dev.to/snyk/how-to-secure-python-flask-applications-2156
The same issue exists in AI-proxy/src/main.rs

Recommendation

***: It is recommended to store the database password securely, such as by using environment variables, a secrets manager.

Client Response

client response : Fixed. Removal of database and database credentials, for replacements and similar have used environment files.

DMC-27:Insecure Node Registration over HTTP

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- ./code/AI-node/main.py#L469

```
469: response = requests.post(register_url, json=data)
```

Description

***: The `register_node` function registers a node using an insecure HTTP connection instead of HTTPS. This is a security issue as it allows an attacker to intercept and manipulate the registration data in transit. The code snippet below illustrates the issue:

```
url = f"http://{connection_host}:{connection_port}"
```

By using `http` instead of `https`, the registration data is sent in plaintext, making it vulnerable to eavesdropping and tampering.

Recommendation

***: To remediate this issue, the `register_node` function should be modified to use HTTPS instead of HTTP. This can be achieved by updating the `url` variable to use the `https` scheme:

```
url = f"https://{connection_host}:{connection_port}"
```

Additionally, it is recommended to verify the identity of the registration server using SSL/TLS certificates to ensure the authenticity of the connection.

Client Response

client response : Fixed. migrated AI proxy to https using TLS certs and made appropriate changes to AI Node. Will Close once committed.

DMC-28:Incorrect Timestamp Comparison Logic Leads to Improper Data Inclusion

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/main.rs#L486-L536

```
async fn get_risk_score_by_timestamp(
    pool: Arc<Pool>,
    time: i64,
) -> MResult<Vec<HashMap<String, String>>> {
    let mut conn = pool.get_conn().await?;
    let query: String = "SELECT wallet, risk_score, transaction_signature, timestamp, signature, node, timeout FROM ip_data".to_string();
    let results_vec: Vec<mysql_async::Row> = conn.query(query).await?;

    let mut results: Vec<HashMap<String, String>> = Vec::new();

    for row in results_vec {
        let wallet: String = row.get("wallet").unwrap();
        let risk_score: f32 = row.get("risk_score").unwrap();
        let transaction_sig: String = row.get("transaction_signature").unwrap();
        let timestamp: String = row.get("timestamp").unwrap();
        let signature: String = row.get("signature").unwrap();
        let node_id: String = row.get("node").unwrap();
        let time_out: i64 = row.get("timeout").unwrap();
        let risk_score_float: f64 = risk_score as f64; // Convert to f64 if it's not already
        let risk_score_str = format!("{:.1}", risk_score_float);
```

```

        let timestamp_as_datetime: DateTime<Utc> = timestamp.parse::<DateTime<Utc>>().unwrap();
        if Utc::now().signed_duration_since(timestamp_as_datetime) < CDuration::seconds(time) {
            let mut entry_map = HashMap::new();
            entry_map.insert("wallet".to_string(), wallet.clone());
            entry_map.insert("risk_score".to_string(), risk_score.to_string());
            entry_map.insert("timeout".to_string(), time_out.to_string());
            entry_map.insert("timestamp".to_string(), timestamp.clone());
            entry_map.insert("signature".to_string(), signature);
            entry_map.insert("node_id".to_string(), node_id);

            let mut data_map = BTreeMap::new();
            data_map.insert("wallet", wallet);
            data_map.insert("risk_score", risk_score_str);
            data_map.insert("transaction_sig", transaction_sig);
            data_map.insert("timestamps", timestamp);
            let data_json = serde_json::to_string(&data_map).unwrap();

            // Turn JSON string into bytes
            let data_bytes = data_json.as_bytes();
```

```
526:
527:             let data_hex = hex::encode(data_bytes);
528:
529:             entry_map.insert("data".to_string(), data_hex);
530:
531:             results.push(entry_map);
532:         }
533:     }
534:
535:     Ok(results)
536: }
```

Description

***: The function `get_risk_score_by_timestamp` in the provided Rust code is intended to filter entries based on their timestamps, specifically to include only those entries that are within a certain time range from the current moment (`Utc::now()`). The critical logic intended to perform this filtering is encapsulated in the following condition:

```
if Utc::now().signed_duration_since(timestamp_as_datetime) < CDuration::seconds(time)
```

This condition aims to include entries where the timestamp is within a specified number of seconds (`time`) from the current time. However, due to a misinterpretation of the `time` parameter, the logic erroneously includes entries that are *older* than the current time minus the specified number of seconds. This misinterpretation results in the function fetching entries from the past up to the present time minus `time` seconds, instead of limiting to recent entries as might be expected by users or specific application contexts.
Since all integers (including `time`) are positive and `CDuration::seconds(time)` generates a positive duration, the condition effectively checks for entries that are older than "now minus `time` seconds." This exposes potentially old, irrelevant, or sensitive data, which could lead to incorrect decision-making or data processing based on an unintended dataset.

Recommendation

***: - **Adjust the condition logic**:

- If the intention is to include data from the recent past up to the current moment:

  ```
  if Utc::now().signed_duration_since(timestamp_as_datetime) <= CDuration::seconds(time)
  ```

- If the intention is to include data from now extending into the future (unlikely but for completeness):

  ```
  if Utc::now().signed_duration_since(timestamp_as_datetime) >= CDuration::seconds(-time)
  ```

Client Response

client response : Fixed. With latest commit and removal of database features, issue has been removed as well.
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-29:Inadequate Handling of Concurrency

Category	Severity	Client Response	Contributor
Race condition	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/ai_node_manager.rs#L14-L63

```
14: pub async fn ai_node_manage(
15:     client: Arc<Client<HttpConnector>>,
16:     ai_node_status: Arc<Mutex<HashMap<String, Status>>>,
17: ) {
18:     loop {
19:         let pending_addresses: Vec<String> = ai_node_status
20:             .lock()
21:             .await
22:             .iter()
23:             .filter_map(|(address, status)| match status {
24:                 Status::Pending(value) if (1..=4).contains(value) => Some(address.clone()),
25:                 _ => None,
26:             })
27:             .collect();
28:
29:         for ai_node_addr in pending_addresses {
30:             let client = Arc::clone(&client);
31:             let shared_ai_node_status = Arc::clone(&ai_node_status);
32:
33:             tokio::spawn(async move {
```

```
34:                 let ai_node_addr_clone = ai_node_addr.clone();
35:
36:                 match check_health(client, ai_node_addr).await {
37:                     Ok(_) => {
38:                         let mut node_status = shared_ai_node_status.lock().await;
39:                         println!("AI node check health ok {:?}", ai_node_addr_clone);
40:                         node_status.insert(ai_node_addr_clone, Status::Active);
41:                     }
42:                     Err(_) => {
43:                         let mut node_status = shared_ai_node_status.lock().await;
44:                         if let Some(status) = node_status.get_mut(&ai_node_addr_clone) {
45:                             if let Status::Pending(value) = status {
46:                                 *value += 1;
47:                                 if *value >= 5 {
48:                                     node_status.insert(ai_node_addr_clone, Status::Inactive);
49:                                 }
50:                             }
51:                         }
52:                     }
53:                 }
```

```
54:             });
55:         }
56:
57:         let mut ai_node_status = ai_node_status.lock().await;
58:         ai_node_status.retain(|_, status| *status != Status::Inactive);
59:         drop(ai_node_status);
60:
61:         sleep(Duration::from_secs(60)).await;
62:     }
63: }
```

Description

***: There is a :

- Potential Deadlock and Resource Exhaustion Due to Inadequate Handling of Concurrency.

The ai_node_manage function in ai_node_manager.rs handles concurrency using `Arc<Mutex<HashMap<String, Status>>>`, which can lead to potential deadlocks and resource exhaustion if not managed carefully. Continuously spawning tasks in a loop without bounds can degrade performance and exhaust system resources.
POC :

```
pub async fn ai_node_manage(
    client: Arc<Client<HttpConnector>>,
    ai_node_status: Arc<Mutex<HashMap<String, Status>>>,
) {
    loop {
        let pending_addresses: Vec<String> = ai_node_status
            .lock()
            .await
            .iter()
            .filter_map(|(address, status)| match status {
                Status::Pending(value) if (1..=4).contains(value) => Some(address.clone()),
                _ => None,
            })
            .collect();
```

```
        for ai_node_addr in pending_addresses {
            let client = Arc::clone(&client);
            let shared_ai_node_status = Arc::clone(&ai_node_status);

            tokio::spawn(async move {
                let ai_node_addr_clone = ai_node_addr.clone();

                match check_health(client, ai_node_addr).await {
                    Ok(_) => {
                        let mut node_status = shared_ai_node_status.lock().await;
                        println!("AI node check health ok {:?}", ai_node_addr_clone);
                        node_status.insert(ai_node_addr_clone, Status::Active);
                    }
                    Err(_) => {
                        let mut node_status = shared_ai_node_status.lock().await;
                        if let Some(status) = node_status.get_mut(&ai_node_addr_clone) {
                            if let Status::Pending(value) = status {
                                *value += 1;
                                if *value >= 5 {
                                    node_status.insert(ai_node_addr_clone, Status::Inactive);
                                }
                            }
                        }
                    }
                }
            });
        }

        let mut ai_node_status = ai_node_status.lock().await;
        ai_node_status.retain(|_, status| *status != Status::Inactive);
        drop(ai_node_status);

        sleep(Duration::from_secs(60)).await;
    }
}
```

Impact:
Deadlock: Improper use of Mutex can lead to deadlocks, causing the application to hang.
Resource Exhaustion: Continuously spawning tasks in a loop without any limit can degrade performance and exhaust system resources.

Recommendation

***: Fix: Use asynchronous data structures to handle concurrency more efficiently and implement rate limiting or backpressure to prevent resource exhaustion.
use tokio::sync::RwLock; // Replace Mutex with RwLock for better concurrency

```
pub async fn ai_node_manage(
    client: Arc<Client<HttpConnector>>,
    ai_node_status: Arc<RwLock<HashMap<String, Status>>>,
) {
    loop {
        let pending_addresses: Vec<String> = {
            let read_guard = ai_node_status.read().await;
            read_guard
                .iter()
                .filter_map(|(address, status)| match status {
                    Status::Pending(value) if (1..=4).contains(value) => Some(address.clone()),
                    _ => None,
                })
                .collect()
        };
```

```
        for ai_node_addr in pending_addresses {
            let client = Arc::clone(&client);
            let shared_ai_node_status = Arc::clone(&ai_node_status);

            tokio::spawn(async move {
                let ai_node_addr_clone = ai_node_addr.clone();

                match check_health(client, ai_node_addr).await {
                    Ok(_) => {
                        let mut write_guard = shared_ai_node_status.write().await;
                        println!("AI node check health ok {:?}", ai_node_addr_clone);
                        write_guard.insert(ai_node_addr_clone, Status::Active);
                    }
                    Err(_) => {
                        let mut write_guard = shared_ai_node_status.write().await;
                        if let Some(status) = write_guard.get_mut(&ai_node_addr_clone) {
                            if let Status::Pending(value) = status {
                                *value += 1;
                                if *value >= 5 {
                                    write_guard.insert(ai_node_addr_clone, Status::Inactive);
                                }
                            }
                        }
                    }
                }
            });
        }

        {
            let mut write_guard = ai_node_status.write().await;
            write_guard.retain(|_, status| *status != Status::Inactive);
        }

        // Sleep with rate limiting to prevent resource exhaustion
        tokio::time::sleep(Duration::from_secs(60)).await;
    }
}
```

By using `RwLock` instead of `Mutex`, we can handle concurrency more efficiently. This allows multiple readers without blocking, and a single writer when necessary. Additionally, implementing rate limiting or backpressure prevents resource exhaustion by controlling the rate of task spawning.

Client Response

client response : Fixed. Implemented recommendation with Rwlock
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-30:Improper Input Validation in Command-Line Argument Parsing

Category	Severity	Client Response	Contributor
Race condition	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/main.rs#L66

```
66: fn parse_cli_arguments() -> (String, String, String) {
```

Description

***: Issue: The current implementation of the parse_cli_arguments function in main.rs does not properly validate command-line arguments. This can lead to invalid addresses being used, which might cause the application to crash or behave unpredictably.
POC :

```
fn parse_cli_arguments() -> (String, String, String) {
    let matches = App::new("ai_proxy")
        .version("0.1.0")
        .author("domichain")
        .about("Create an HTTP JSON-RPC proxy to forward 'sendTransaction' requests to an AI node.")
        .arg(
            Arg::with_name("proxy_addr")
                .short("p")
                .long("proxy_addr")
                .value_name("Proxy Address")
                .required(true)
                .help("Sets proxy address"),
        )
        .arg(
            Arg::with_name("register_addr")
                .short("r")
                .long("register_addr")
                .value_name("Register Address")
                .required(true)
                .help("Sets Register Address"),
        )
        .arg(
            Arg::with_name("domi_node_addr")
                .short("n")
                .long("domi_node_addr")
                .value_name("Domi Node Address")
                .required(true)
                .help("Sets domi node address"),
        )
        .get_matches();
    (
        matches.value_of("proxy_addr").unwrap().to_string(),
        matches.value_of("register_addr").unwrap().to_string(),
        matches.value_of("domi_node_addr").unwrap().to_string(),
    )
}
```

Improper input validation can lead to the application accepting invalid or malicious inputs, which can cause the application to crash or behave unpredictably. This can be exploited by attackers to perform denial-of-service attacks or other malicious activities.

Recommendation

***: Fix: Implement thorough validation of command-line arguments to ensure they are in the correct format and within expected ranges. Handle potential errors gracefully to avoid panics.

```
fn parse_cli_arguments() -> Result<(String, String, String), &'static str> {
    let matches = App::new("ai_proxy")
        .version("0.1.0")
        .author("domichain")
        .about("Create an HTTP JSON-RPC proxy to forward 'sendTransaction' requests to an AI node.")
        .arg(
            Arg::with_name("proxy_addr")
                .short("p")
                .long("proxy_addr")
                .value_name("Proxy Address")
                .required(true)
                .help("Sets proxy address"),
        )
        .arg(
            Arg::with_name("register_addr")
                .short("r")
                .long("register_addr")
                .value_name("Register Address")
                .required(true)
                .help("Sets Register Address"),
        )
        .arg(
            Arg::with_name("domi_node_addr")
                .short("n")
                .long("domi_node_addr")
                .value_name("Domi Node Address")
                .required(true)
                .help("Sets domi node address"),
        )
        .get_matches();

    let proxy_addr = matches.value_of("proxy_addr").ok_or("Missing proxy address")?;
    let register_addr = matches.value_of("register_addr").ok_or("Missing register address")?;
    let domi_node_addr = matches.value_of("domi_node_addr").ok_or("Missing domi node address")?;

    // Add more validation logic here if necessary
    Ok((proxy_addr.to_string(), register_addr.to_string(), domi_node_addr.to_string()))
}
```

By adding proper validation and handling potential errors, we can ensure that the command-line arguments are correctly formatted and within expected ranges, preventing potential crashes or unpredictable behavior caused by invalid inputs.

Client Response

client response : Fixed. Implemented a validation check for each input with the following function fn validate_address(address: &str) -> Result<(), Box> { let re = Regex::new(r"^(?:[0-9]{1,3}.){3}[0-9]{1,3}:\d+$")?; if !re.is_match(address) { return Err(format!("Invalid address format: {}", address).into()); } Ok(()) }

Commit link: https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-31:If there are no active nodes, all transactions are silently ignored

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/AI-proxy/src/ai_handler.rs#L174-L224

```
174: let active_addresses: Vec<String> = ai_node_status
175:                 .iter()
176:                 .filter(|(_, status)| **status == Status::Active)
177:                 .map(|(address, _)| address.clone())
178:                 .collect();
179:
180:             // Release the lock to avoid holding it while performing async operations
181:             drop(ai_node_status);
182:
183:             // Check if ai_trans_count is 30
184:             if ai_trans_count >= 30 {
185:                 println!("1...sending {} tx to ai node", ai_trans_count);
186:
187:                 let ai_transactions_list_clone = ai_transactions_list.clone();
188:                 let mut ai_request = AIRequest::default();
189:                 ai_request.transactions = ai_transactions_list_clone;
190:                 let serialized_ai_request = ai_request
191:                     .write_to_bytes()
192:                     .expect("Failed to serialize message");
193:                 let shared_serialized_ai_request = Arc::new(serialized_ai_request);
```

```
194:                 for ai_node_addr in active_addresses {
195:                     let client = Arc::clone(&client);
196:                     let data = shared_serialized_ai_request.clone();
197:                     let shared_ai_node_status_clone = Arc::clone(&shared_ai_node_status);
198:
199:                     tokio::spawn(async move {
200:                         let ai_node_addr_clone = ai_node_addr.clone();
201:                         println!(
202:                             "2...sending {} tx to ai node {:?}",
203:                             ai_trans_count, ai_node_addr_clone
204:                         );
205:                         match send_to_ai_node(client, data.to_vec(), ai_node_addr).await {
206:                             Ok(_) => {
207:                                 println!(
208:                                     "3....sending {} tx to ai node {:?}",
209:                                     ai_trans_count, ai_node_addr_clone
210:                                 );
211:                             }
212:                             Err(_) => {
213:                                 // Acquire a lock on shared_ai_node_status_clone
```

```
                                let mut ai_node_status = shared_ai_node_status_clone.lock().await;
                                ai_node_status.insert(ai_node_addr_clone, Status::Pending(1));
                            }
                        }
                    });
                }

                ai_trans_count = 0;
                ai_transactions_list.clear();
            }
        }
```

Description

***: In `ai_handle.rs`, the `transaction_handler` will query all active nodes:

```
// Collect active addresses
            let active_addresses: Vec<String> = ai_node_status
                .iter()
                .filter(|(_, status)| **status == Status::Active)
                .map(|(address, _)| address.clone())
                .collect();
```

and then use these nodes to handle transactions:

```
for ai_node_addr in active_addresses {
                    let client = Arc::clone(&client);
                    let data = shared_serialized_ai_request.clone();
                    let shared_ai_node_status_clone = Arc::clone(&shared_ai_node_status);

                    tokio::spawn(async move {
                        let ai_node_addr_clone = ai_node_addr.clone();
                        println!(
                            "2...sending {} tx to ai node {:?}",
                            ai_trans_count, ai_node_addr_clone
                        );
                        match send_to_ai_node(client, data.to_vec(), ai_node_addr).await {
                            Ok(_) => {
                                println!(
                                    "3....sending {} tx to ai node {:?}",
                                    ai_trans_count, ai_node_addr_clone
                                );
                            }
                            Err(_) => {
                                // Acquire a lock on shared_ai_node_status_clone
                                let mut ai_node_status = shared_ai_node_status_clone.lock().await;
                                ai_node_status.insert(ai_node_addr_clone, Status::Pending(1));
                            }
                        }
                    });
                }
```

At last, these transactions will be cleared:

```
ai_trans_count = 0;
ai_transactions_list.clear();
```

The issue here is that if there are no active nodes, all transactions are silently ignored. Users will mistakenly think that transactions have been executed, however in reality these transactions are simply discarded.

Recommendation

***: Consider returning error when there are no active nodes.

Client Response

client response : Acknowledged.
Secure3: . changed severity to Informational

DMC-32:Cursor not properly closed in database operation

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/AI-node/main.py#L71-L222

```
71: def get_risk_score_by_timestamp(time):
72:
73:     try:
74:         with getConnection() as conn:
75:             cursor = conn.cursor()
76:             query = """
77:                 SELECT wallet, risk_score, transaction_signature, timestamp, signature, node, timeout
78:                 FROM ip_data
79:             """
80:             cursor.execute(query)
81:             rows = cursor.fetchall()
82:
83:             results = []
84:
85:             for row in rows:
86:                 wallet = row[0]
87:                 risk_score = row[1]
88:                 transaction_sig = row[2]
89:                 timestamps = row[3]
90:                 signature = row[4]
```

```
91:                 node_id = row[5]
92:                 time_out = row[6]
93:
94:                 data_dict = {
95:                     "wallet": wallet,
96:                     "risk_score": str(risk_score),
97:                     "transaction_sig": transaction_sig,
98:                     "timestamps": timestamps
99:                 }
100:
101:                 data_bytes = json.dumps(data_dict).encode()
102:                 data = data_bytes.hex()
103:
104:                 # Check if there are timestamps and if the latest timestamp is within the time range
105:                 if datetime.fromisoformat(timestamps) >= \
106:                         (datetime.now(timezone.utc) - timedelta(seconds=int(time))):
107:
108:                     select_query = "SELECT public_key FROM node_public_keys WHERE node_id = %s"
109:                     cursor.execute(select_query, (node_id,))
110:                     row = cursor.fetchone()
```

```
                    public_key = row[0]

                    # Append the result as a dictionary to the list
                    result = {
                        'risk_score': risk_score,
                        'wallet': wallet,
                        'data': data,
                        'timeout': time_out,
                        'timestamp': timestamps,
                        'public_key': public_key,
                        'signature': signature
                    }
                    results.append(result)

            return results
    except Error as e:
        print("Error retrieving data from the database:", str(e))
        return []


```

```
def create_table():
    try:
        with getConnection() as conn:
            cursor = conn.cursor()
            cursor.execute('''CREATE TABLE IF NOT EXISTS ip_data (
                                wallet VARCHAR(255),
                                risk_score FLOAT,
                                ip_address VARCHAR(255),
                                transaction_signature VARCHAR(255),
                                timestamp VARCHAR(255),
                                signature VARCHAR(255),
                                node VARCHAR(255),
                                timeout VARCHAR(255)
                            )''')

            # Create the Account ID table
            cursor.execute('''CREATE TABLE IF NOT EXISTS node_public_keys (
                                node_id INT AUTO_INCREMENT PRIMARY KEY,
                                public_key VARCHAR(255)
                            )''')
```

```
151:     except Error as e:
152:         print("Error creating table:", str(e))
153:
154:
155: create_table()
156:
157:
158: def insert_node_public_key(public_key):
159:     try:
160:         with getConnection() as conn:
161:             cursor = conn.cursor()
162:             insert_query = "INSERT INTO node_public_keys (public_key) VALUES (%s)"
163:             cursor.execute(insert_query, (public_key,))
164:             conn.commit()
165:     except Error as e:
166:         print("Error inserting node public key:", str(e))
167:
168:
169: def insert_public_key_from_env():
170:     public_key = os.getenv("PUBLIC_KEY")
```

```
171:
172:     if public_key:
173:         try:
174:             with getConnection() as conn:
175:                 cursor = conn.cursor()
176:                 # Check if the public key is already in the table
177:                 select_query = "SELECT node_id FROM node_public_keys WHERE public_key = %s"
178:                 cursor.execute(select_query, (public_key,))
179:                 row = cursor.fetchone()
180:
181:                 if row:
182:                     # Public key already exists in the table
183:                     account_id = row[0]
184:                   #  print("Public key already exists in the table. Node ID:", account_id)
185:                 else:
186:                     # Insert the public key into the table
187:                     insert_node_public_key(public_key)
188:                     cursor.execute(select_query, (public_key,))
189:                     account_id = cursor.fetchone()[0]
190:                    # print("Public key inserted. New Node ID:", account_id)
```

```
191:
192:                 global node_id
193:                 node_id = account_id
194:         except Exception as e:
195:             print("Error inserting/public key from .env:", str(e))
196:             sys.exit(1)
197:     else:
198:         print("No public key found in .env file")
199:         sys.exit(1)
200:
201:
202: def update_database(wallet, risk_score, ip_address, transaction_signatures, timestamps, signature, accoun
t_id, timeout):
203:     try:
204:         with getConnection() as conn:
205:             cursor = conn.cursor()
206:
207:             # Encode Signature
208:             signature_hex = signature.hex()
209:
210:             # Insert a new record
```

```
211:             insert_query = """
212:                 INSERT INTO ip_data (wallet, risk_score, ip_address, transaction_signature,
213:                 timestamp, signature, node, timeout)
214:                 VALUES (%s, %s, %s, %s, %s, %s, %s, %s)
215:             """
216:             cursor.execute(insert_query, (wallet, risk_score, ip_address, transaction_signatures,
217:                                           timestamps, signature_hex, account_id, timeout))
218:
219:             conn.commit()
220:            # print("Inserted new entry into DB")
221:     except Exception as e:
222:         print("Error inserting into database:", str(e))
```

Description

***: - The cursor is not explicitly closed after use. Although the connection is managed with a context manager (`with getConnection() as conn`), the cursor should also be explicitly closed to ensure all resources are released properly.

- Not closing the cursor can lead to resource leaks, which might exhaust the database resources, causing the application to crash or behave unexpectedly under load.

Recommendation

***: Use a context manager (`with` statement) to ensure the cursor is properly closed after its use.

```
def create_table():
    try:
        with getConnection() as conn:
            with conn.cursor() as cursor:
                # ...
    except Error as e:
        print("Error creating table:", str(e))
```

Client Response

client response : Fixed. Removal of database in AI Proxy and Node removes this issue with the database
https://github.com/Domino-Blockchain/AI-proxy/commit/3151420f6d05fc5cee5f7edac891475b1336e1de

DMC-33:Cloning using arc multiple times in the loop in `ai_node_manage()` unnecessarily hikes up cost

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	***

Code Reference

- code/AI-proxy/src/ai_node_manager.rs#L30-L31

```
30: let client = Arc::clone(&client);
31:             let shared_ai_node_status = Arc::clone(&ai_node_status);
```

Description

***:
Take a look at https://github.com/Secure3Audit/code_DomiChain_AI/blob/e1498d6b77df311d0da558fcde77e1249fd3134c/code/AI-proxy/src/ai_node_manager.rs#L30-L31

```
            let client = Arc::clone(&client);
            let shared_ai_node_status = Arc::clone(&ai_node_status);
```

The code currently clones the `client` (an `Arc<Client<HttpConnector>>`) and `ai_node_status` (an `Arc<Mutex<HashMap<String, Status>>>`) inside the loop using `Arc::clone(&client)` and `Arc::clone(&ai_node_status)`.
But this concept seems to be unnecessary imo

- `Arc` is a Rust construct designed for thread-safe reference counting. It allows multiple threads to share ownership of a single underlying object, which is considerably costly to use.
- In this case, both `client` and `ai_node_status` are already wrapped in `Arc`, indicating they are intended for concurrent access.
- Cloning them within the loop creates unnecessary overhead, as each iteration allocates a new `Arc` object even though they all reference the same underlying data.

Impact:

- This repeated cloning can lead to performance degradation, especially when dealing with a large number of pending addresses.
- The memory overhead of additional `Arc` objects may also be a concern in resource-constrained environments.

Recommendation

***:
Consider moving the cloning of `client` and `ai_node_status` outside the loop. Perform the cloning once before the loop starts, and then capture references to the cloned objects within the loop iterations.

Client Response

client response : Fixed.

DMC-34:Base64 malleable risk

Category	Severity	Client Response	Contributor
Language Specific	Informational	Acknowledged	***

Code Reference

- code/AI-proxy/src/ai_handler.rs#L102-L105

```
102: TransactionEncoding::Base64 => match base64::decode(tx_str) {
103:                     Ok(decoded_bytes) => Ok(decoded_bytes),
104:                     Err(err) => Err(format!("Error decoding base64: {err:?}")),
105:                 },
```

Description

***: Base64 is not a rigorous serialization algorithm and is not suitable for use in blockchain systems.
You can find the potential decoding attack for Base64 here: https://eprint.iacr.org/2022/361.pdf

Recommendation

***: Consider using Base58 only.

Client Response

client response : Acknowledged. Validator and blockchain uses base58, while the proxy has base64 decoding compatibility it is not used to store and information or data, likely can be removed entirely as base58 is the only one used - if this is the case will update with the fixed label.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.