Home assignment - Technical Support Manager @Bloxroute

By Mauro Monso

As part of the recruitment process for the Technical Support Manager role at BloXroute I have been given a home assignment, this is the deliverable.

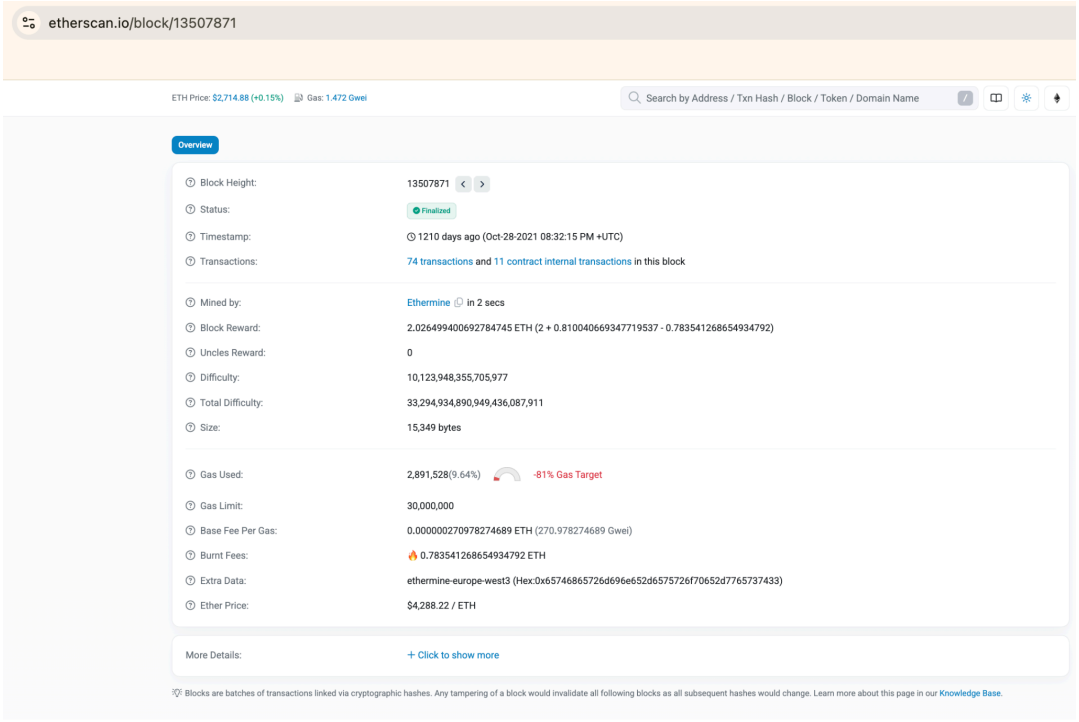
Exercise 1

For this exercise please use Etherscan(https://etherscan.io/) for inspection at first and then write scripts to query data.

[Etherscan Inspection]

Using Etherscan, find block 13507871 and provide the following information:

In this case we searched and found the required block on etherscan:



1. Who mined the block?

The block was mined by mining pool Ethermine with address 0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8

2. How many transactions does it contain?

There are 74 transactions and 11 contract internal transactions in this block.
A transaction is a cryptographically signed instruction that changes the blockchain state.
Contract Internal Transactions are the result of contract execution on the Ethereum Blockchain.

3. Which transaction is sent by `Coinbase 1` account

(0x71660c4005ba85c37ccec55d0c4493e66fe775d3)?

The Transaction that happened from the Wallet Coinbase 1, with TxHash is a Transfer function for 108 USDC (Circle Stablecoin) to the Wallet address 0x3Be960FEF469FEe13A34A475f5b57E9ec2A98998

[Script 1]

Please focus on the transaction sent by `Coinbase 1` account above. You may need a free Ethereum node endpoint from https://infura.io/ or from https://www.alchemy.com/. You may also find web3 library useful, for example, web3.js in nodejs and web3.py in python.

Write a script to use the endpoint to get the following information about this transaction:

1). To address

2). Gas price

3). Input call data

Answer

For this task, I have written a Python Script, the code is visible here as Script1a.py

To run it it is needed:

Python installed
An Infura account. Please replace <YOUR_PROJECT_ID> within the code with your actual value.
The Web3.py library (pip install web3).

Script Steps:

Connect to Ethereum via Infura
Fetches transaction for the specified Tx Hash (in this example it is hardcoded the Tx Hash for the transaction sent by Coinbase 1) details like:
To Address
Gas Price
Input Call Data

Running the script:

This is the terminal when running the script:

```
chimuel@Chimuels-MacBook-Air Bloxroute % python3 Script1a.py
1) To Address: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48
2) Gas Price: 272.978274689 Gwei
3) Input Call Data:
   - Function Selector: 0xa9059cbb
   - Recipient: 0x3Be960FEF469FEe13A34A475f5b57E9ec2A98998
   - Amount: 108.0 USDC
```

The script matches in its input what can be seen in the Input Data Decoder and Etherscan for the TxHash.

[Script 1b]

Say now you are the owner of this `Coinbase 1` account, and you would like to transfer 100 USDC tokens `Binance 10` account (0x85b931A32a0725Be14285B66f1a22178c672d69B). Please construct the input call data.

Answer

If we go back to the Input Data Decoder for the previous transaction and we want to create the transaction with the new parameters we press the “copy all parameters” button, we see:

```
{
  "function": "transfer(address,uint256)",
  "params": [
    "0x3Be960FEF469FEe13A34A475f5b57E9ec2A98998",
    "108000000"
  ]
}
```

If we modify it accordingly, we have:

```
{
  "function": "transfer(address,uint256)",
  "params": [
    "0x85b931A32a0725Be14285B66f1a22178c672d69B",
    "100000000"
  ]
}
```

Moreover, I have created a script to showcase the input data with the required parameters, the code is visible here in Script1b.py.

Terminal:

```
chimuel@Chimuels-MacBook-Air Bloxroute % python3 Script1b.py
Encoded Transfer Call Data:
   - Function Selector: a9059cbb
   - Recipient: 0x85b931A32a0725Be14285B66f1a22178c672d69B
   - Amount: 100.0 USDC
   - Input Call Data:
0xa9059cbb00000000000000000000000085b931a32a0725be14285b66f1a22178c672d69b0000000000000000000000000000000000000000000000000000005f5e100
```

[Etherscan Inspection]

In Ethereum, blocks are mined by different miners, and miners are usually able to mine more than one block every day. For the blocks after 13507871, find out the next block (at block height h) mined by this miner.

The next block mined by Ethermine, address 0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8 is 13507875.

[Script 2]

Please focus on this block at block height h, write a script to use the endpoint to query this block, and get the following information:

1). Sender (from address) that sent most transactions in this block

2). Receiver (to address) that received most transactions in this block

3). Transaction with the highest gas price

Answer

For this task I have created another script, the code is in Script1c.py, the block number 13507875 is hardcoded.

The output for this script is:

```
chimuel@Chimuels-MacBook-Air Bloxroute % python3 Script1c.py
1) Sender with most transactions:
0x46340b20830761efd32832A74d7169B29FEB9758 (16 txns)
2) Receiver with most transactions:
0xA090e606E30bD747d4E6245a1517EbE430F0057e (98 txns)
3) Transaction with highest gas price:
   - Hash:
dbac7f98d43874d28c8cff965acda3ddf6a868dba2df4ac618b451a5c8361654
   - From: 0x8Fe748F51D4c7893BE9Ee177091e8e8e20Af3029
   - To: 0xe66B31678d6C16E9ebf358268a790B763C133750
   - Gas Price: 324 Gwei
   - Value: 0.08195819195545221 ETH
```

Exercise 2

1.A customer is experiencing an issue with submitting Ethereum transactions, and is claiming that the transaction was not processed correctly because the transaction cannot be found on Etherscan block explorer. How would you troubleshoot the issue and respond to the customer? What if this happens on Solana network, how would the troubleshooting process be different?

Answer

When a customer reports that their **Ethereum** transaction is missing on **Etherscan** or a similar block explorer, follow these structured troubleshooting steps. If the issue occurs on the **Solana** network, the process differs due to its unique architecture.

Ethereum Troubleshooting Steps

1. Verify Transaction Hash

- Confirm the customer has the correct transaction hash.
- Check the hash on Etherscan to see if it appears.

2. Check for Network Delays or Reorgs

- Network congestion or chain reorganizations can delay or drop transactions.
- Use Etherscan's gas tracker to see if the gas price was too low.

3. Check Mempool and see if it is a pending transaction.

The transaction might still be in the pending pool (mempool) and hasn't been mined yet.

I have created a script to check if given a Tx Hash, the transaction has been mined or not, the code can be found in Script2a.py.

Example:

I ran the script for for TxHash "0x21ed4bd5b1bf07851bdaf981318ba3f3269db100eab5ee0b10c66aea1d527ee4"which was hardcoded.

Results:

```
chimuel@Chimuels-MacBook-Air Bloxroute % python3 Script2a.py

Transaction has been mined.

Block Number: 13507871
```

```
Gas Used: 21000
```

4. Check for Gas Price Issues

- If the transaction is stuck in pending due to low gas fees, the customer can:
    - Replace the transaction using the same nonce but with a higher gas price.
    - Cancel the transaction by submitting a 0 ETH transaction to themselves with the same nonce but a higher fee.

5. Finalize the Solution

- If the transaction eventually confirms: Direct the customer to view it on Etherscan.
- If the transaction is dropped: Recommend resubmitting it with optimized gas fees.

Solana Troubleshooting Steps

Solana's consensus mechanism (Proof of History + Tower BFT) and transaction handling differ from Ethereum, so the troubleshooting approach is adapted.

To troubleshoot the issue of a missing Solana transaction on a block explorer, follow this structured approach:

1. Confirm Transaction Details

First, verify the transaction details provided by the customer:

- Confirm the transaction signature is correct.
- Ask for additional details such as the sending wallet address, receiving wallet address, and the timestamp of the transaction.

2. Check Solana Network Status

Network congestion or outages can cause delays:

- Visit Solana Status to check if there are any network issues.
- Confirm that the block explorer is operational.

3. Check Transaction on Solana Explorer

Search for the transaction using the official Solana block explorer: Solscan or Solana Explorer.

- Enter the transaction signature to see if it appears.
- If not found, proceed to the next step.

4. Query the Solana RPC Endpoint

I have developed a Python script to check the status of the transaction, Script2Solana.py. You can run it, you need to put the Tx Signature in the script to search for it.

I have tested with a signature example

- Transaction Found: If the transaction is found, analyze the status:
    - Check for "meta": { "err": null } to confirm success.
    - If "err" is not null, it indicates a failure (e.g., insufficient funds, invalid nonce).
- Transaction Not Found: If the transaction is not found:
    - The transaction might not have been broadcasted properly.
    - It could also be due to network delays or dropped transactions.

5. Troubleshoot Potential Issues

If the transaction is not found:

- Network Congestion: Confirm if the network is congested (using Solana Status).
- Dropped Transaction: The transaction might have been dropped due to insufficient fees or other network issues.
- Resend Transaction: Advise the customer to attempt resubmitting the transaction with a higher fee.

2. Many API endpoints are executed in the form of HTTP requests. Suppose an inexperienced programmer writes a piece of code (which might be a small part of a more complicated program) in a programming language like javascript or python, to invoke some API endpoints. This programmer receives an error message “AxiosError: Request failed with status code 500” in their javascript code, and they think the API endpoints are not working properly. How would you work with this inexperienced programmer to troubleshoot the problem?

Answer

To help an inexperienced programmer troubleshoot the error “AxiosError: Request failed with status code 500” in their JavaScript code, I would follow these steps:

1. Explain the Error Code:

- A 500 status code indicates a server-side error, meaning something went wrong on the server while processing the request.
- It's not necessarily a problem with the API endpoint itself; it could be related to the data sent, the request format, or server conditions.

2. Review the Code:

- Ask the programmer to share the relevant code snippet where they are making the API request using Axios.
- Specifically, look for:

    - The URL being called.
    - The HTTP method used (GET, POST, PUT, DELETE, etc.).
    - Any headers set in the request (e.g., authentication tokens, content type).
    - The data being sent in the request (especially for POST or PUT requests).

3. Add Error Handling:

- Implement proper error handling to get more details about the error:

```
axios.post('https://example.com/api', { data: 'example' })
  .then(response => {
    console.log('Success:', response.data);
  })
  .catch(error => {
    if (error.response) {
      // Server responded with a status code outside 2xx
      console.log('Error Status:', error.response.status);
      console.log('Error Data:', error.response.data);
      console.log('Error Headers:', error.response.headers);
    } else if (error.request) {
      // Request was made but no response was received
      console.log('No response received:', error.request);
    } else {
      // Something else caused the error
      console.log('Error Message:', error.message);
    }
  });
```

- This will provide more context on why the server is responding with a 500 error.

4. Check the API Endpoint:

- Verify if the API endpoint is correct and functioning by testing it in a tool like Postman or cURL.
- Check if the API has documentation specifying required parameters, headers, or authentication methods.

5. Validate Request Data:

- Make sure the payload being sent to the server is in the correct format (e.g., JSON, form data).
- Double-check for missing or incorrect fields in the request data.

6. Review Server Logs (If Possible):

- If the programmer has access to server logs, review them to find more specific error messages.

- If not, ask the API provider for additional insights into the server error.

7. Test with Minimal Request:

- Simplify the request to the bare minimum and gradually add more complexity to identify the breaking point.

```
axios.get('https://example.com/api/test')
  .then(response => console.log(response.data))
  .catch(error => console.log('Error:', error.message));
```

8. Common Issues to Check:

- Authentication Issues: Incorrect or missing API keys, tokens, or headers.
- Data Validation: The server may be expecting certain fields or specific data formats.
- CORS Issues: If running in a browser, ensure the server supports Cross-Origin Requests.
- Rate Limiting: Some APIs limit the number of requests in a certain period.

9. Encourage Debugging Mindset:

- Guide the programmer to break down the problem systematically.
- Emphasize the importance of reading error messages carefully and using console logs for debugging.

By following this structured approach, the inexperienced programmer will not only resolve the specific issue but also gain valuable troubleshooting skills for future development challenges.

3. An inexperienced programmer just got the latest Apple MacBook with M4 Chip, and wanted to run some blockchain node clients. On this new laptop, the programmer first attempted to run go ethereum client with docker image `ethereum/client-go:v1.14.7` (https://hub.docker.com/layers/ethereum/client-go/v1.14.7/images/sha256-6f3391660ae655e46 bc2ddb53e03f9a70e92c99ab1ae63d681db408fcd8efb45), and things appear to be working okay, later attempted to run solana client docker image `solanalabs/solana:v1.18.26` (https://hub.docker.com/layers/solanalabs/solana/v1.18.26/images/sha256-098806e64d44bccdb edbf07c2edabd1c850b92c8a4a0f81eba9c789034813db6) released by solana foundation (https://github.com/solana-labs/solana/tree/master/sdk/docker-solana). Will the solana docker container start properly? What potential error/warning message may show up, and why is that? Please focus on computer architecture in this question.

Answer

The Ethereum Docker container supports ARM64 Architecture. However, the Solana Docker container is likely to fail to start properly on the new MacBook with the M4 chip, and here's why:

1. Architecture Compatibility Issue

- The Apple M4 chip is based on the ARM architecture (ARM64).
- The Solana Docker image (`solanalabs/solana:v1.18.26`) is built for the x86_64 architecture (common for Intel and AMD processors).
- Docker on macOS can emulate x86_64 architecture using qemu, but this may not always work smoothly, especially for high-performance applications like blockchain nodes.

2. Potential Error/Warning Messages
You may see errors such as:

```
standard_init_linux.go:219: exec user process caused: exec format error
```

- This occurs when the executable inside the Docker container is for x86_64 but is being run on ARM64.

Alternatively, you might see a warning related to platform mismatch:

```
WARNING: The requested image's platform (linux/amd64) does not match the detected host platform (linux/arm64/v8) and no specific platform was requested
```

- Docker is detecting the ARM architecture but the image is built for AMD64.

3. Why This Happens

- Binary Incompatibility: The binaries inside the Solana Docker image are compiled for x86_64, which is not natively compatible with ARM64.
- Emulation Limitations: While Docker Desktop on Mac with M1/M2/M4 chips supports emulation using qemu, it may not be reliable for all applications, leading to crashes or poor performance.

4. Possible Solutions

- Use an ARM-Compatible Image: If available, use a Docker image that is built for ARM64. You could check if Solana Labs provides one or build it yourself from source.

  ```
  docker pull solanalabs/solana:v1.18.26 --platform linux/arm64
  ```

- Build from Source: Compile the Solana binaries directly on the M4 MacBook to ensure native ARM compatibility.
- Enable Emulation: You can try forcing emulation, but expect performance overhead: `docker run --platform linux/amd64 solanalabs/solana:v1.18.26`

Summary

The Solana Docker container may not start properly due to an architecture mismatch between the ARM64-based M4 chip and the x86_64 binaries inside the Docker image. The most reliable solution would be to find or build an ARM-compatible version of the Solana node client.

4. In a blockchain network, validators are responsible for producing blocks and propagating them to the rest of the network. Consider a scenario where one validator is in-turn to produce the next block, this validator finds that the block is expected to contain a transaction paid to the validator wallet itself sent by other random people. The validator wallet's balance is expected to increase after the execution of this transaction, so the validator duplicated the same transaction multiple times, and added all of them to the next block, in order to receive the same transfer multiple times. This validator then finishes construction of the block and propagates the block to other nodes in the network. What would happen next to the transactions (and that block)? Will this validator wallet balance receive the duplicated transfers?

Answer

No, the validator's wallet balance will **not** receive the duplicated transfers, and the block containing the duplicated transactions will likely be **rejected** by the network. Here's why:

1. Transaction Uniqueness and Nonce Checking

In blockchain networks like Ethereum or other EVM-compatible chains:

- Each transaction has a nonce, which is a unique number associated with the sender's wallet. The nonce is incremented for every new transaction made by that wallet.
- If the same transaction is duplicated, the nonce would also be duplicated, which would make the transactions invalid. The network nodes will detect this and reject the block.

2. Block Validation by Other Nodes

- When the block is propagated, other validators and nodes in the network will verify the block's validity before accepting it.
- During this verification, they will check for:
    - Double-spending: The same transaction cannot be executed twice.
    - Nonce uniqueness: As mentioned, nonces cannot be duplicated.

    - Signature validity: The transactions must be signed by the sender's private key, and if duplicated, the signatures will also be identical, raising a red flag.

3. Consensus Mechanism Reaction

- In Proof of Stake (PoS) or Proof of Work (PoW) blockchains:
    - If the block is found to contain invalid transactions, it will be rejected by the majority of the nodes.
    - The network will ignore this block, and the next validator in line will be chosen to produce a new block.
    - The malicious validator may also face penalties, such as losing their stake or being temporarily banned, depending on the blockchain's governance rules.

4. No Wallet Balance Increase

Since the block will be rejected:

- The duplicated transactions will not be added to the blockchain.
- The validator's wallet balance will not receive any of the duplicated transfers.

5. Summary

- The block with duplicated transactions will be rejected during the validation phase.
- The validator's wallet balance will not increase from the duplicated transactions.
- The validator might face penalties for attempting this malicious behavior.

This mechanism ensures the integrity and security of the blockchain network by preventing double-spending and maintaining transaction uniqueness.

5. Normally when one transfers a token to a random address, the token is considered as lost and cannot be recovered. This is usually true in a decentralized network like Bitcoin and Ethereum. For example, one transfers 100 USDT tokens to zero address https://etherscan.io/address/0x0000000000000000000000000000000000000000 on Ethereum Mainnet by mistake, the 100 USDT tokens are lost. Suppose you can now control all the ethereum validators, would you be able to get 100 USDT tokens back, and how? USDT is issued by Tether, suppose you are now the owner of Tether, would you be able to get 100 USDT tokens back, and how?

Answer

1. If You Control All Ethereum Validators:

If you control all Ethereum validators, theoretically, you could recover the 100 USDT tokens by performing the following steps:

- Fork the Blockchain: You could propose a hard fork to reverse the transaction. This would involve rewriting the transaction history so that the 100 USDT are not sent to the zero address. This is similar to what happened with the DAO hack rollback on Ethereum.
- State Change: Directly modify the state of the blockchain to reflect that the 100 USDT are back in the sender's wallet. This would require consensus from all validators, which you control in this scenario.
- Risks and Consequences: Doing so would compromise the integrity and trust of the blockchain as it would prove that the network is centralized and transactions can be reversed. This would be highly controversial.

2. If You Are the Owner of Tether (USDT Issuer):

As the owner of Tether, you have more practical and less controversial options:

- Blacklisting and Reissuing Tokens:
    - Blacklisting: Tether has the power to blacklist specific addresses. This means they can freeze the 100 USDT that were sent to the zero address, rendering them unusable.
    - Reissuing: After blacklisting the lost tokens, Tether can mint new 100 USDT tokens and send them to the original owner, effectively restoring the lost amount.
    - Example: Tether has previously blacklisted addresses as seen on Etherscan using their centralized authority over USDT smart contracts.

Conclusion:

- With control over validators: It is theoretically possible but goes against the principles of blockchain immutability and decentralization.
- As Tether owner: It is more straightforward, as Tether has centralized control over USDT tokens and can blacklist and reissue tokens easily.

In practice, Tether's centralized control makes them the only entity realistically capable of recovering the lost tokens without undermining blockchain principles.

Exercise 3

[Optional]

Please send a code example for a smart contract that performs the following:

1. Track and display the amount of ETH and any ERC20 token this contract holds

2. Allow any sender to send ETH to this contract.

3. Define a function that allows the contract owner to withdraw all available ETH stored in this contract.

4. The function defined above should also allow any (non-owner) wallet to withdraw back 10% of the ETH stored in this contract.

If you deploy the smart contract to a testnet please verify it on etherscan and send the details.

Answer

I have deployed a Solidity smart contract on Sepolia testnet that does the following:

1. Tracks and displays the amount of ETH and any ERC20 tokens it holds.
2. Allows anyone to send ETH to the contract.
3. Allows the contract owner to withdraw all available ETH.
4. Allows any non-owner wallet to withdraw 10% of the ETH stored in the contract.

The code is available in Optional.sol.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

interface IERC20 {
    function balanceOf(address account) external view returns (uint256);
    function transfer(address recipient, uint256 amount) external returns (bool);
}

contract OptionalEthAndTokenTracker {
    address public owner;
    mapping(address => bool) public hasWithdrawn;
    uint256 public totalWithdrawals;

    constructor() {
        owner = msg.sender;
    }

    // Fallback and Receive Functions to accept ETH
    receive() external payable {}
    fallback() external payable {}

    // Function to get the ETH balance of this contract
    function getEthBalance() public view returns (uint256) {
        return address(this).balance;
    }

    // Function to get the ERC20 token balance of this contract
    function getTokenBalance(address tokenAddress) public view returns (uint256) {
        return IERC20(tokenAddress).balanceOf(address(this));
    }

    // Function to withdraw all ETH (Owner only)
    function withdrawAllEth() public {
        require(msg.sender == owner, "Only owner can withdraw all ETH");
        uint256 amount = address(this).balance;
        payable(owner).transfer(amount);
    }

    // Function for non-owners to withdraw 10% of the ETH balance
    function withdrawTenPercent() public {
        require(msg.sender != owner, "Owner cannot withdraw 10%");
        require(!hasWithdrawn[msg.sender], "You have already withdrawn your 10%");
        uint256 tenPercent = address(this).balance / 10;
        require(tenPercent > 0, "Insufficient balance to withdraw 10%");
```

```
        hasWithdrawn[msg.sender] = true;
        totalWithdrawals += tenPercent;
        payable(msg.sender).transfer(tenPercent);
    }
}
```

Key Features Explained

1. Track ETH and ERC20 Token Balances:
    - getEthBalance() returns the ETH balance of the contract.
    - getTokenBalance(address tokenAddress) returns the balance of any ERC20 token held by the contract.
2. Receive ETH:
    - The contract can receive ETH via receive() and fallback() functions.
3. Owner Withdrawal:
    - withdrawAllEth() allows only the contract owner to withdraw all available ETH.
4. Non-Owner Withdrawal (10% Rule):
    - withdrawTenPercent() allows non-owners to withdraw 10% of the ETH balance, but only once per wallet.

For testing, I have sent some USDC Test tokens, whose contract address is: 0x13fA158A117b93C27c55b8216806294a0aE88b6D.

Contract Token Owner: 0x81E8Ab9b7D9Ad67F9d6cd30B2B396d7A806DCE0a

On Sepolia Etherscan it is visible several tests that were performed over the contract.