Assure DeFi®

THE VERIFICATION GOLD STANDARD

VERIFIED BY ASSURE DEFI ® INTEGRITY ✦ TRUST ✦ CREDIBILITY

Security Assessment

Safesun

Date: 20/08/2024

Audit Status: FAIL

Audit Edition: Advanced

ASSURE DEFI®

THE VERIFICATION GOLD STANDARD

Risk Analysis

Vulnerability summary

Classification	Description
🔴 High	High-level vulnerabilities can result in the loss of assets or manipulation of data.
🟠 Medium	Medium-level vulnerabilities can be challenging to exploit, but they still have a considerable impact on smart contract execution, such as allowing public access to critical functions.
🟡 Low	Low-level vulnerabilities are primarily associated with outdated or unused code snippets that generally do not significantly impact execution, sometimes they can be ignored.
🟢 Informational	Informational vulnerabilities, code style violations, and informational statements do not affect smart contract execution and can typically be disregarded.

Executive Summary

According to the Assure assessment, the Customer's smart contract is **Poorly Secured.**

Insecure	**Poorly Secured**	Secured	Well Secured

Scope

Target Code And Revision

For this audit, we performed research, investigation, and review of the Safesun contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

Target Code And Revision

Project	Assure
Language	Solidity
Codebase	Safesun-sniper-bot-main.zip [SHA256] 35cdcf9396e0c14f7d587a9c7f1ee1154f7cc147a7e6d33fce9bd33e487a0b2c
Audit Methodology	Static, Manual

Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

Category	Item
Code review & Functional Review	● Compiler warnings. ● Race conditions and Reentrancy. Cross-function race conditions. ● Possible delays in data delivery. ● Oracle calls. ● Front running. ● Timestamp dependence. ● Integer Overflow and Underflow. ● DoS with Revert. ● DoS with block gas limit. ● Methods execution permissions. ● Economy model. ● Private user data leaks. ● Malicious Event log. ● Scoping and Declarations. ● Uninitialized storage pointers. ● Arithmetic accuracy. ● Design Logic. ● Cross-function race conditions. ● Safe Zeppelin module. ● Fallback function security. ● Overpowered functions / Owner privileges

AUDIT OVERVIEW

HIGH

1. Title: Plaintext Storage of Private Keys in User Accounts

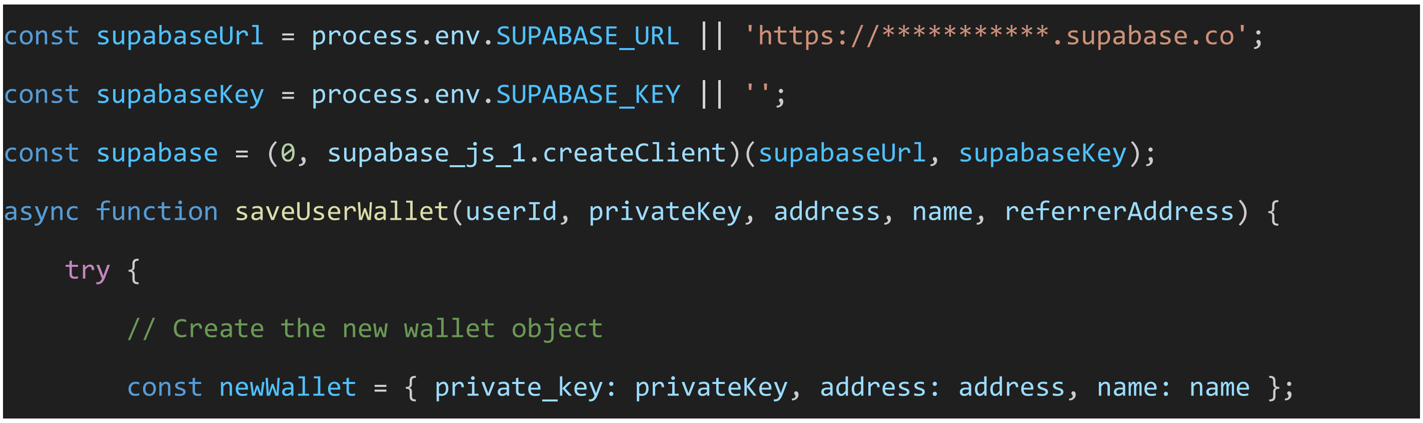
Issue:

Although Tron Manager follows good security practices for generating the admin account via environment variables, the private keys of user accounts are stored in plaintext in a centralized database (SUPABASE). This presents a significant security risk as user accounts should not be accessible by a centralized entity and/or the private keys should be encrypted, for instance, using AES-256, before being stored.

Fix:

Private keys must be encrypted before storage and [**The encryption keys themselves should be stored securely, ideally in an environment that supports secure key management (e.g., AWS KMS, Azure Key Vault, or Google Cloud KMS*).], and efforts should be made to reduce the centralization of the system wherever possible.

Code:



```
const supabaseUrl = process.env.SUPABASE_URL || 'https://***********.supabase.co';
const supabaseKey = process.env.SUPABASE_KEY || '';
const supabase = (0, supabase_js_1.createClient)(supabaseUrl, supabaseKey);
async function saveUserWallet(userId, privateKey, address, name, referrerAddress) {
    try {
        // Create the new wallet object
        const newWallet = { private_key: privateKey, address: address, name: name };
```

MEDIUM

No medium severity issues were found.

LOW

No low severity issues were found.

INFORMATIONAL

No informational severity issues were found.

Testing ‘Mocks’

During the testing phase, custom use cases were written to cover all the logic of the typescript bot.

Check “Annexes” to see the testing code.

Safesun Mocks:

ApiKeyRotator:

```
export class ApiKeyRotator {
    private apiKeys: string[];
    private currentIndex: number;

    constructor(apiKeys: string[]) {
      this.apiKeys = apiKeys;
      this.currentIndex = 0;
    }

    getNextKey(): string {
      const key = this.apiKeys[this.currentIndex];
      this.currentIndex = (this.currentIndex + 1) % this.apiKeys.length;
      return key;
    }
  }
```

TronWebManager:

```
// TronWeb Mock
const mockTronWeb = {
  trx: {
    getBalance: jest.fn().mockResolvedValue((10 + 1.1) * 1e6), // Simulates a balance in TRX
    getTransaction: jest.fn().mockResolvedValue({ ret: [{ contractRet: 'SUCCESS' }] }) // Simulates a confirmed transaction
  },
  contract: jest.fn().mockReturnValue({
    swapExactETHForTokens: jest.fn().mockReturnValue({
      send: jest.fn().mockResolvedValue({ txID: 'mockTxId' })
    }),
    balanceOf: jest.fn().mockResolvedValue(BigInt(1000)), // Simulates a token balance
    approve: jest.fn().mockReturnValue({
      send: jest.fn().mockResolvedValue({ txID: 'mockApprovalTxId' })
    }),
    swapExactTokensForETH: jest.fn().mockReturnValue({
      send: jest.fn().mockResolvedValue({ txID: 'mockSwapTxId' })
    }),
    getAmountsOut: jest.fn().mockResolvedValue({ amounts: [0, 1000] })
  }),
  setHeader: jest.fn() // Mocks setHeader
};

// tronWebManager Mock
const tronWebManager = {
  getTronWebInstance: jest.fn().mockImplementation((privateKey: string) => {
    if (!privateKey || privateKey.length < 64) {
      throw new Error('Invalid private key provided'); // Throws error if the key is invalid
    }
    return mockTronWeb;
  }),
  rotateApiKey: jest.fn()
};

export default tronWebManager;
```

Utils:

```
const utils = {
    address: {
        toHex: jest.fn()
    }
};

export default utils;
```

Annexes

Testing code:

BuyTokentest:

```
import { Wallet, buyToken } from '../blockchain/sunswap';
import pumpSwapAbi from "../abis/PumpSwapAbi";
import tronWebManager from './__mocks__/tronWebManager';
import utils from './__mocks__/utils';

// Mock of tronWeb and contracts
const mockTronWeb = {
    trx: {
      getBalance: jest.fn(),
      getTransaction: jest.fn()
    },
    contract: jest.fn().mockReturnValue({
      swapExactETHForTokens: jest.fn().mockReturnValue({
        send: jest.fn().mockResolvedValue({ txID: 'mockTxId' })
      }),
      balanceOf: jest.fn().mockResolvedValue(BigInt(1000)),
      approve: jest.fn().mockReturnValue({
        send: jest.fn().mockResolvedValue({ txID: 'mockApprovalTxId' })
      }),
      swapExactTokensForETH: jest.fn().mockReturnValue({
        send: jest.fn().mockResolvedValue({ txID: 'mockSwapTxId' })

      }),

      getAmountsOut: jest.fn().mockResolvedValue({ amounts: [0, 1000] })

    }),

    setHeader: jest.fn()

  };



const mockWallet: Wallet = {

    user_id: 1,

    private_key: 'mockPrivateKey',

    address: 'mockAddress'

};



const SUNSWAP_CONTRACT_ADDRESS = 'mockContractAddress';

const WTRX = 'mockWTRX';



describe('buyToken', () => {

    beforeEach(() => {

      tronWebManager.getTronWebInstance = jest.fn().mockReturnValue(mockTronWeb);

      utils.address.toHex.mockReturnValue('mockFormattedAddress');

      process.env.TRON_API_KEYS = 'mock_key_1,mock_key_2';

    });



    it('should send a transaction and poll for confirmation successfully', async () => {

      mockTronWeb.trx.getBalance.mockResolvedValue((10 + 1.1) * 1e6);

      const pollForConfirmation = jest.fn().mockResolvedValue('mockTransaction');

      await expect(buyToken(mockWallet, 'mockTokenAddress',
10)).resolves.toBe('mockTransaction');
      expect(mockTronWeb.contract).toHaveBeenCalledWith(pumpSwapAbi,
SUNSWAP_CONTRACT_ADDRESS);
      expect(mockTronWeb.trx.getBalance).toHaveBeenCalledWith(mockWallet.address);
    });

    it('should throw an error if the balance is insufficient', async () => {
      mockTronWeb.trx.getBalance.mockResolvedValue(5 * 1e6);

      await expect(buyToken(mockWallet, 'mockTokenAddress', 10)).rejects.toThrow();
    });

    it('should handle errors from tronWeb.getTransaction', async () => {
      mockTronWeb.trx.getBalance.mockResolvedValue((10 + 1.1) * 1e6);
      jest.spyOn(global.console, 'error').mockImplementation(() => {});

      const pollForConfirmation = jest.fn().mockRejectedValue(new Error('Polling error'));

      await expect(buyToken(mockWallet, 'mockTokenAddress', 10)).rejects.toThrow('Polling
error');
      expect(console.error).toHaveBeenCalledWith('Error in buyToken function:', new
Error('Polling error'));
    });

    it('should call rotateApiKey multiple times', async () => {
      mockTronWeb.trx.getBalance.mockResolvedValue((10 + 1.1) * 1e6);

      const pollForConfirmation = jest.fn().mockResolvedValue('mockTransaction');

      await buyToken(mockWallet, 'mockTokenAddress', 10);

      expect(tronWebManager.rotateApiKey).toHaveBeenCalledTimes(2); // Called before and after transaction
    });
  });
```

PollForConfirmationTests:

```
import { pollForConfirmation } from '../blockchain/sunswap';
import tronWebManager from './__mocks__/tronWebManager';

const mockTronWeb = {
  trx: {
    getTransaction: jest.fn()
  }
};

describe('pollForConfirmation', () => {
  beforeEach(() => {
    tronWebManager.rotateApiKey.mockReturnValue(undefined);
  });
  it('should confirm transaction within max attempts', async () => {
    mockTronWeb.trx.getTransaction.mockResolvedValue({ ret: [{ contractRet: 'SUCCESS' }] });

    await expect(pollForConfirmation(mockTronWeb, 'mockTxId')).resolves.toEqual({ ret: [{ contractRet: 'SUCCESS' }] });
  });

  it('should throw an error if transaction is not confirmed', async () => {
    mockTronWeb.trx.getTransaction.mockResolvedValue({ ret: [{ contractRet: 'FAILED' }] });

    await expect(pollForConfirmation(mockTronWeb, 'mockTxId')).rejects.toThrow('Transaction confirmation timed out');
  });
});
```

SellTokensTests:

```
import { Wallet, sellToken } from '../blockchain/sunswap';
import tronWebManager from './__mocks__/tronWebManager';
import utils from './__mocks__/utils';

// Set up mocks
const mockTronWeb = {
    trx: {
      getBalance: jest.fn(),
      getTransaction: jest.fn()

    },

    contract: jest.fn().mockReturnValue({

      swapExactETHForTokens: jest.fn().mockReturnValue({

        send: jest.fn().mockResolvedValue({ txID: 'mockTxId' })

      }),

      balanceOf: jest.fn().mockResolvedValue(BigInt(1000)),

      approve: jest.fn().mockReturnValue({

        send: jest.fn().mockResolvedValue({ txID: 'mockApprovalTxId' })

      }),

      swapExactTokensForETH: jest.fn().mockReturnValue({

        send: jest.fn().mockResolvedValue({ txID: 'mockSwapTxId' })

      }),

      getAmountsOut: jest.fn().mockResolvedValue({ amounts: [0, 1000] })

    }),

    setHeader: jest.fn()

  };


const mockWallet: Wallet = {

  user_id: 1,

  private_key: 'valid_private_key_which_is_64_characters_long',

  address: 'mockAddress'

};


const SUNSWAP_CONTRACT_ADDRESS = 'mockContractAddress';

const WTRX = 'mockWTRX';
```

```
describe('sellToken', () => {
  beforeEach(() => {
    process.env.TRON_API_KEYS = 'mock_key_1,mock_key_2';
    tronWebManager.getTronWebInstance = jest.fn().mockReturnValue(mockTronWeb);
    utils.address.toHex.mockReturnValue('mockFormattedAddress');
  });

  it('should approve and swap tokens and poll for confirmation successfully', async () => {
    const pollForConfirmation = jest.fn().mockResolvedValue('mockSwapTxId');

    await expect(sellToken(mockWallet, 'mockTokenAddress', '1000')).resolves.toBe('mockSwapTxId');
    expect(mockTronWeb.contract).toHaveBeenCalled();
    expect(mockTronWeb.contract().approve).toHaveBeenCalledWith(SUNSWAP_CONTRACT_ADDRESS, '1000');
    expect(mockTronWeb.contract().swapExactTokensForETH).toHaveBeenCalled();
  });

  it('should throw an error if token balance is insufficient', async () => {
    mockTronWeb.contract().balanceOf.mockResolvedValue(BigInt(500));

    await expect(sellToken(mockWallet, 'mockTokenAddress', '1000')).rejects.toThrow('Insufficient token balance');
  });

  it('should handle errors from token approval or swap transaction', async () => {
    jest.spyOn(global.console, 'error').mockImplementation(() => {});

    mockTronWeb.contract().approve.mockReturnValue({
      send: jest.fn().mockRejectedValue(new Error('Approval error'))
    });

    await expect(sellToken(mockWallet, 'mockTokenAddress',
'1000')).rejects.toThrow('Approval error');
    expect(console.error).toHaveBeenCalledWith('Error in sellToken function:', new
Error('Approval error'));
  });

  it('should calculate minimum output amount correctly', async () => {
    const amountIn = '1000';
    const amountOut = 1000;
    const slippage = 0.96; // 4% slippage
    const minOutput = (BigInt(amountOut) * BigInt(96)) / BigInt(100);

    mockTronWeb.contract().getAmountsOut.mockResolvedValue({ amounts: [amountIn, amountOut]
});

    await sellToken(mockWallet, 'mockTokenAddress', amountIn);

    expect(mockTronWeb.contract().swapExactTokensForETH).toHaveBeenCalledWith(
      amountIn,
      minOutput.toString(),
      [ 'mockTokenAddress', WTRX ],
      'mockFormattedAddress',

      expect.any(Number)

    );

  });



  it('should call rotateApiKey multiple times', async () => {

    const pollForConfirmation = jest.fn().mockResolvedValue('mockSwapTxId');



    await sellToken(mockWallet, 'mockTokenAddress', '1000');



    expect(tronWebManager.rotateApiKey).toHaveBeenCalledTimes(3); // Called for approval, swap, and confirmation

  });

});
```

Technical Findings Summary

Findings

Vulnerability Level	Total	Pending	Not Apply	Acknowledged	Partially Fixed	Fixed
🔴 High	1					
🟠 Medium	0					
🟡 Low	0					
🟢 Informational	0					

Assessment Results

Score Results

Review	Score
Global Score	**75/100**
Assure KYC	Not completed
Audit Score	75/100

The Following Score System Has been Added to this page to help understand the value of the audit, the maximum score is 100, however to attain that value the project must pass and provide all the data needed for the assessment. Our Passing Score has been changed to 84 Points for a higher standard, if a project does not attain 85% is an automatic failure. Read our notes and final assessment below. The Global Score is a combination of the evaluations obtained between having or not having KYC and the type of contract audited together with its manual audit.

Audit FAIL

Following our comprehensive security audit of the staking contract for Safesun project, The recent security audit has unfortunately resulted in a failure. Critical vulnerabilities were identified that pose significant risks to the integrity and security of the system. These issues must be addressed promptly to ensure compliance with industry standards and to protect against potential threats. A detailed report of the findings has been provided, outlining the necessary steps to remediate the identified weaknesses. Immediate action is required to rectify these issues and to schedule a follow-up audit to confirm the effectiveness of the implemented fixes.

Disclaimer

Assure Defi has conducted an independent security assessment to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the reviewed code for the scope of this assessment. This report does not constitute agreement, acceptance, or advocating for the Project, and users relying on this report should not consider this as having any merit for financial advice in any shape, form, or nature. The contracts audited do not account for any economic developments that the Project in question may pursue, and the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude, and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are entirely free of exploits, bugs, vulnerabilities or deprecation of technologies.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence, regardless of the findings presented. Information is provided ‘as is, and Assure Defi is under no covenant to audit completeness, accuracy, or solidity of the contracts. In no event will Assure Defi or its partners, employees, agents, or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions or actions with regards to the information provided in this audit report.

The assessment services provided by Assure Defi are subject to dependencies and are under continuing development. You agree that your access or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies with high levels of technical risk and uncertainty. The assessment reports could include false positives, negatives, and unpredictable results. The services may access, and depend upon, multiple layers of third parties.