hashlock.

Security Audit

MiracleDigithread (Wallet)

September 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit Scope	7
Security Rating	8
Intended Smart Contract Functions	9
Code Quality	10
Audit Resources	10
Dependencies	10
Severity Definitions	11
Status Definitions	12
Audit Findings	13
Centralisation	37
Conclusion	38
Our Methodology	39
Disclaimers	41
About Hashlock	42

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

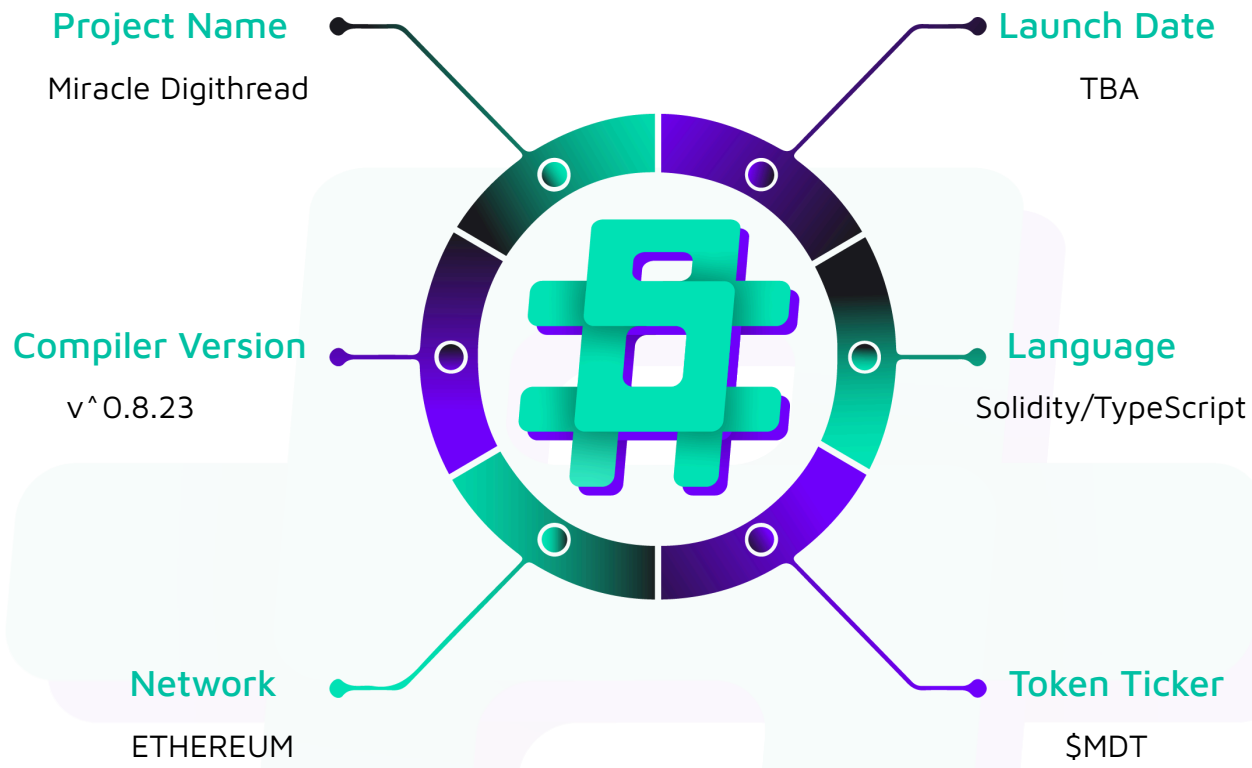
The Miracle Digithread team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

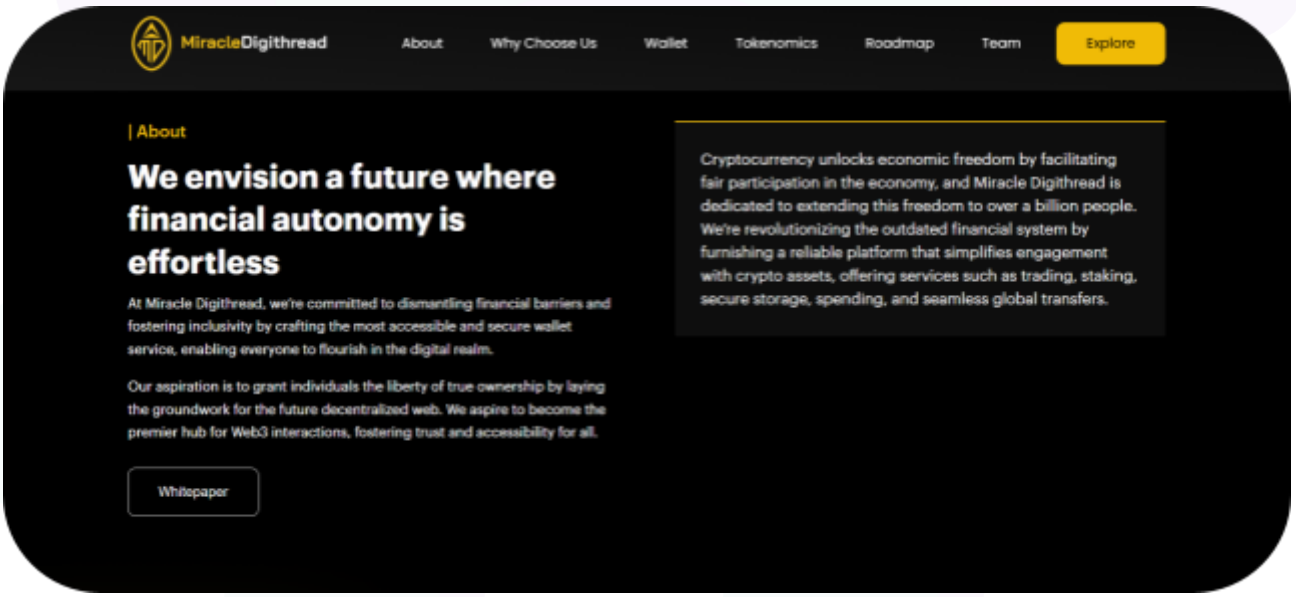
MiracleDigithread (MDT) is a next-generation Web3 ecosystem designed to empower users with true ownership of their digital assets through a self-custodial wallet, a BEP-20 native token, and a full suite of decentralized finance tools. Built on BNB Chain for speed and low fees, MDT bridges crypto utility with real-world adoption by combining staking rewards, seamless global transfers, and a crypto card solution that lets holders spend their tokens anywhere. With its transparent tokenomics, audited smart contracts, and roadmap toward launching its own blockchain, MiracleDigithread positions itself as more than just a token—it's an entire digital economy where decentralization, transparency, and usability converge. By integrating wallet flexibility, mobile apps, staking, and future blockchain infrastructure, MDT aims to drive mass adoption of DeFi while giving users full control, security, and real utility for their assets.

Project Name: Miracle Digithread
Project Type: Wallet, DeFi
Compiler Version: ^0.8.23
Website: https://miracledigithread.com/
Logo:

Visualised Context:



Project Visuals:



Audit Scope

We at Hashlock audited the solidity code within the Miracle Digithread project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Miracle Digithread Smart Contracts**
Platform	**Ethereum / Solidity / TypeScript**
Audit Date	**September, 2025**
Contract 1	hardhat.config.js
Contract 2	DigiToken.sol
Contract 3	deploy.js
Contract 4	forking.js
Contract 5	verify.js
Audited GitHub Commit Hash	056c1ea5909bed60eebab6948b61df25598278ac
Fix Review GitHub Commit Hash	647660fbe8317f60fe939b46c860d91f74382e20

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved or acknowledged.

Hashlock found:

1 High severity vulnerabilities

1 Medium severity vulnerabilities

3 Low severity vulnerabilities

2 Gas Optimisations

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
DigiToken.sol - Hold and Transfer ERC20 Tokens -Freeze and Unfreeze - Token Burning (Global) - Claim Rewards (For users)	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Miracle Digithread project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring were recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Miracle Digithread project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] DigiToken#stopFreezing - Anyone (Except Admin) Can Permanently Halt Freezing (DoS)

Description

The `stopFreezing` function implements inverted access control: it reverts when the caller is the admin and allows any non-admin to execute. As a result, any external account (other than admin) can disable freezing globally by setting `freezingAllowed = false`. Because there is no complementary function to re-enable freezing, this creates a permanent denial-of-service for new freezes. The issue is triggered immediately after deployment and under all conditions.

Vulnerability Details

The function checks if (`msg.sender == admin`) revert `NotAdmin()`, which is the logical opposite of the intended admin-only restriction. Thus, only the admin is blocked, while all non-admins are authorized.

```
function stopFreezing() external {

    if (msg.sender == admin) {

        revert NotAdmin();

    }

    freezingAllowed = false;

    emit freezingStopped(msg.sender);

}
```

Once called, freeze becomes permanently unusable because it guards on `freezingAllowed`:

```
function freeze(uint _amount, uint duration) external freezingLimitNotReached {

    if (freezingAllowed == false) {

        revert FreezingStopped();

    }

    ...

}
```

There is no function to restore `freezingAllowed` to true, so a single non-admin transaction can irreversibly disable the product's core feature.

Proof of Concept

```
function testStopFreezingAccessControl() public {
    // --- Setup and Preconditions ---
    // Ensure DigiToken is initialized and 'alice' and 'owner' addresses are available

    // --- Exploit Execution ---
    // 'alice' (not admin) calls stopFreezing, should succeed
    vm.startPrank(alice);
    digiToken.stopFreezing();
    vm.stopPrank();

    // 'owner' (admin) calls stopFreezing, should revert with "NotAdmin()"
    vm.startPrank(owner);
    vm.expectRevert("NotAdmin()");
    digiToken.stopFreezing();
    vm.stopPrank();

    // --- Post-condition Check ---
    // No further assertions needed; PoC succeeds if only non-admin can call stopFreezing
    assertTrue(true);
    // PoC succeeds if only non-admin can call stopFreezing
}
```

Impact

Any non-admin address can permanently disable new freezes for all users, causing a protocol-wide denial of service. Since there is no mechanism to resume freezing, the disruption is irreversible without an upgrade. This directly harms user experience, halts growth, and can invalidate economic assumptions tied to freeze inflows.

Recommendation

Enforce proper authorization and consider adding a symmetric "start" function. Minimal fixes:

```
function stopFreezing() external {
   if (msg.sender == admin) {
        revert NotAdmin();
   }
   freezingAllowed = false;
   emit freezingStopped(msg.sender);
}
function stopFreezing() external {
    if (msg.sender != admin) revert NotAdmin();
    freezingAllowed = false;
    emit freezingStopped(msg.sender);
}

function startFreezing() external {
    if (msg.sender != admin) revert NotAdmin();
    freezingAllowed = true;
}
```

Status

Resolved

Medium

[M-01] DigiToken#freeze - Incorrect Threshold Condition Prevents Price Updates

Description

The `freeze` function is responsible for pushing new timestamps into priceChangeTime once certain thresholds of `toDtalFreezed` are reached. However, the condition only checks for `priceChangeTime.length == 2` and `priceChangeTime.length == 3`. This means that when `priceChangeTime.length` is 0 or 1, the thresholds for 810,000 MDT or 1,620,000 MDT are never triggered, and the array never updates. As a result, the pricing mechanism relying on `priceChangeTime` becomes stuck at the base value.

Vulnerability Details

The following code shows the problematic logic:

```
// @note : intended to push timestamps at freeze thresholds
if (priceChangeTime.length == 2 && totalFreezed >= 810000 * (10 ** decimals()))
    priceChangeTime.push(block.timestamp);
else if (priceChangeTime.length == 3 && totalFreezed >= 1620000 * (10 ** decimals()))
    priceChangeTime.push(block.timestamp);
```

Because the initial `priceChangeTime.length` is 0, the first threshold (810,000 MDT) is never reached in practice, and no push occurs. This prevents `getPrice()` from ever progressing past its initial pricing state, since that function bases its logic on `priceChangeTime.length`. As a result, prices remain at 20 even after large amounts of tokens have been frozen.

Proof of Concept

```
function testPriceDoesNotChangeWithFreeze() public {
    // --- Setup and Preconditions ---
    // Calculate the total amount to be distributed to 'alice'
    uint256 totalAmount = 1_000_000e18;
    // Calculate the freeze amount per iteration based on the current price
    uint256 amount = (10_000 * 1e18) / digiToken.getPrice(block.timestamp);
```

```
    // Allocate the total amount to 'alice'
    deal(address(digiToken), alice, totalAmount);

    // --- Exploit Execution ---
    // Repeatedly freeze tokens in small increments to surpass the threshold
    for (uint i = 0; i < totalAmount / amount; i++) {
        vm.startPrank(alice);
        digiToken.freeze(amount, 12);
        vm.stopPrank();
    }

    // --- Post-condition Check ---
    // Check that the totalFreezed value exceeds the threshold
      assertGt(digiToken.totalFreezed(), 810_000e18); // PoC succeeds if totalFreezed > 810,000 MDT
    // Check that the price does not change as expected due to the flawed condition
     assertEq(digiToken.getPrice(block.timestamp), 20); // PoC succeeds if price remains 20
}
```

Impact

The pricing mechanism fails to evolve as intended, leaving the token stuck at an artificially low base price. This breaks economic assumptions, prevents accurate price progression, and allows users to freeze large amounts of tokens without ever incurring the expected higher price tiers. Over time, this undermines fairness and system design assumptions.

Recommendation

The condition should handle the case where `priceChangeTime.length == 0` or `1`. A safer approach is to check thresholds directly against `totalFreezed`, rather than relying solely on array length.

Status

Acknowledged

Low

[L-01] DigiToken - Use of Floating Solidity Pragma

Description

The contract specifies a floating pragma ^0.8.23, which allows compilation with any version of Solidity 0.8.23 or higher within the 0.8.x range. This introduces risk because different compiler versions may have distinct behaviors, gas optimizations, or even security-related bugs. For financial contracts handling user funds, relying on a floating pragma undermines reproducibility and audit reliability.

Recommendation

Pin the Solidity compiler to an exact version to ensure consistent bytecode and avoid accidental use of a compiler version with undiscovered issues. For example:

```
- pragma solidity ^0.8.23;
+ pragma solidity 0.8.23;
```

This guarantees that all builds use the same verified compiler version.

Status

Acknowledged

[L-02] DigiToken - Inconsistent Use of msg.sender Instead of `_msgSender()`

Description

The contract directly references msg.sender in multiple functions instead of using `_msgSender` from `ContextUpgradeable`. Since `OwnableUpgradeable` relies on `ContextUpgradeable`, this inconsistency introduces potential incompatibilities with meta-transactions or relayer-based calls. While not an immediate exploitable bug, it reduces maintainability and could break expected behavior in certain deployment environments.

Recommendation

For consistency and future compatibility with meta-transactions and upgradeable patterns, replace all `msg.sender` references with `_msgSender()`. Example:

```
- if (_freezedetail.user != msg.sender) {
+ if (_freezedetail.user != _msgSender()) {
      revert NotTheUser();
  }
```

Apply this change across all functions (`transfer`, `transferFrom`, `burn`, `freeze`, `claimReward`, `unfreezeAmount`, etc.).

Status

Resolved

[L-03] DigiToken - Unprotected Implementation Initialization Enables Hostile Ownership

Description

The upgradeable implementation contract can be initialized directly by any external account because `initialize` is exposed with the initializer modifier but there is no constructor guard (e.g., `_disableInitializers`), allowing first-caller ownership capture. After taking ownership, the attacker can invoke `onlyOwner` functions on the implementation.

Vulnerability Details

The implementation's `initialize` is callable on the implementation address itself and confers ownership to the caller:

```
function initialize(
    address _admin,
    address[] memory wallets,
    uint256[] memory amounts
) external initializer {
    if (_admin == address(0)) revert ZeroAddress();
    if (wallets.length != amounts.length || wallets.length != 10) revert InvalidInput();
    __Ownable_init(_admin);
    __ERC20_init("MiracleDigiThread", "MDT");
    admin = _admin;
    transferOwnership(admin); // Ownership gets assigned to the provided _admin
    ...
}
```

Because the implementation lacks a constructor that disables initializers, anyone can call initialize on the implementation address to set themselves (or any address they control) as owner.

Proof of Concept

```
function testImplementationContractUnprotectedInit() public {
```

```
// --- Setup and Preconditions ---
// Start impersonating the 'hacker' address
vm.startPrank(hacker);

// Prepare arrays of wallets and corresponding token amounts for initialization
address[] memory wallets = new address[](10);
wallets[0] = 0x03bAe882d52ac5F370bBA3FB0Aa415DA9e578E4e;
wallets[1] = 0xF66Bc3aD8f6DcfF417a83F3a8d62F5fc5eb70c4e;
wallets[2] = 0x80968767a9bbfDd95857e4E438fc39FDE1ACD4Ca;
wallets[3] = 0x72679dAbF86A46B7b315C3B8040D8f09f2d3c7b8;
wallets[4] = 0xaC0F4aA1F5D6A19dFc1b391B8D4e6345475f1269;
wallets[5] = 0xEb544Bf152fc10C96EA918B5B12F9080dE16910d;
wallets[6] = 0xd1A5d65f191432Dc0B354206B62786D2799E4dad;
wallets[7] = 0xE8e37B6d2281C68D8eA50Cd6BA7e78A964276A69;
wallets[8] = 0x0d35E2Daeb02309f7a57536E7f125d5A9c14f850;
wallets[9] = 0x0491C6776b1FC5e509e85A4B7164C37f6211DE56;

uint256[] memory amounts = new uint256[](10);
amounts[0] = 135000000000000000000000000;
amounts[1] = 270000000000000000000000000;
amounts[2] = 162000000000000000000000000;
amounts[3] = 135000000000000000000000000;
amounts[4] = 135000000000000000000000000;
amounts[5] = 108000000000000000000000000;
amounts[6] = 135000000000000000000000000;
amounts[7] = 135000000000000000000000000;
amounts[8] = 216000000000000000000000000;
amounts[9] = 540000000000000000000000000;

// --- Exploit Execution ---
// Call initialize on the implementation contract directly as 'hacker'
DigiToken(tokenImpl).initialize(hacker, wallets, amounts);

// --- Post-condition Check ---
```

```
    // Confirm that 'hacker' is now the owner of the implementation contract
    assertEq(DigiToken(tokenImpl).owner(), hacker); // PoC succeeds if hacker is owner

    // Call destroyContract to selfdestruct the implementation contract
    DigiToken(tokenImpl).destroyContract(payable(hacker));
}
```

Impact

An attacker can permanently seize ownership of the implementation and exercise `onlyOwner` capabilities (e.g., destroyContract).

Recommendation

Prefer OZ's pattern: `_disableInitializers()` in the implementation constructor so only the proxy can be initialized.

```
contract DigiToken is OwnableUpgradeable, ERC20Upgradeable {

+   constructor() {
+       _disableInitializers(); // Prevent initializing the implementation directly
+   }

   function initialize(
       address _admin,
       address[] memory wallets,
       uint256[] memory amounts
   ) external initializer {
       ...
       __Ownable_init(_admin);
       __ERC20_init("MiracleDigiThread", "MDT");
       admin = _admin;
       transferOwnership(admin);
       ...
   }
```

Status

Acknowledged

Gas

[G-01] DigiToken - Redundant Balance Checks Increase Gas

Description

Both `transfer` and `transferFrom` perform explicit balance checks (`balanceOf(...) < value`) before invoking the `ERC20` transfer logic. In OpenZeppelin's `ERC20Upgradeable`, the internal accounting reverts on underflow, making these pre-checks redundant. These extra `SLOADs` and conditionals add unnecessary gas to every transfer path without improving safety.

Recommendation

Remove the redundant `balanceOf` checks and rely on the inherited `ERC20` logic to revert on insufficient balance. This saves one or more `SLOADs` and a conditional branch per call.

```
function transfer(address to, uint256 value) public override returns (bool) {
-   if (balanceOf(msg.sender) < value) {
-       revert NotEnoughBalance();
-   }
    if (balanceOf(msg.sender) - value < freezedAmount[msg.sender]) {
        revert FreezedTokenTransfer();
    }
    _transfer(msg.sender, to, value);
    return true;
}



function transferFrom(address from, address to, uint256 value) public override returns
(bool) {
-   if (balanceOf(from) < value) {
-       revert NotEnoughBalance();
-   }
    if (balanceOf(from) - value < freezedAmount[from]) {
        revert FreezedTokenTransfer();
```

```
    }
    address spender = _msgSender();
    _spendAllowance(from, spender, value);
    _transfer(from, to, value);
    return true;
}
```

Status

Resolved

[G-02] DigiToken#getPrice - Repeated SLOAD of priceChangeTime.length Increases Gas

Description

`getPrice` reads `priceChangeTime.length` multiple times, causing repeated `SLOADs`. Since `priceChangeTime` is a storage `array, each .length` access incurs a storage read. Caching the length into a local uint len (memory) reduces gas by avoiding duplicate storage reads and branch re-evaluations. This function can be called frequently by other logic, so the micro-optimization is worthwhile.

Recommendation

Cache the array length once and branch on the cached value. This removes repeated `SLOADs` and slightly shortens branching.

```
function getPrice(uint time) public view returns (uint price) {
-    if(priceChangeTime.length == 0) price = 20;
-    else if(priceChangeTime.length == 1 || priceChangeTime.length == 2) price = 40;
-    else if (priceChangeTime.length == 3) price = 60;
-    else if (priceChangeTime.length == 4) price = 80;
-    else price = 80;
+    uint256 len = priceChangeTime.length;
+    if (len == 0) price = 20;
+    else if (len == 1 || len == 2) price = 40;
+    else if (len == 3) price = 60;
+    else if (len == 4) price = 80;
+    else price = 80;
}
```

Status

Resolved

Centralisation

The Miracle Digithread project values decentralisation alongside security and utility.

The protocol's functions are designed to operate independently, minimising reliance on the internal team and hardcoded values, while maintaining robust security and functionality.

Ownership and Upgradeability Note

Following the completion of the audit for the Miracle Digithread smart contract system, the project team has renounced ownership of the Proxy Contract. The ownership renunciation transaction can be verified onto the Blockchain (BSC).

As a result, the proxy is now non-upgradeable and no administrative privileges remain with any external wallet or deployer address. This ensures that the deployed logic contracts are now immutable and the Miracle Digithread system operates in a fully decentralized and trustless manner post-audit.

Conclusion

After Hashlock's analysis, the Miracle Digithread project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved or acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au