hashlock.

Security Audit

N4T (dApp)

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

Table of Contents

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

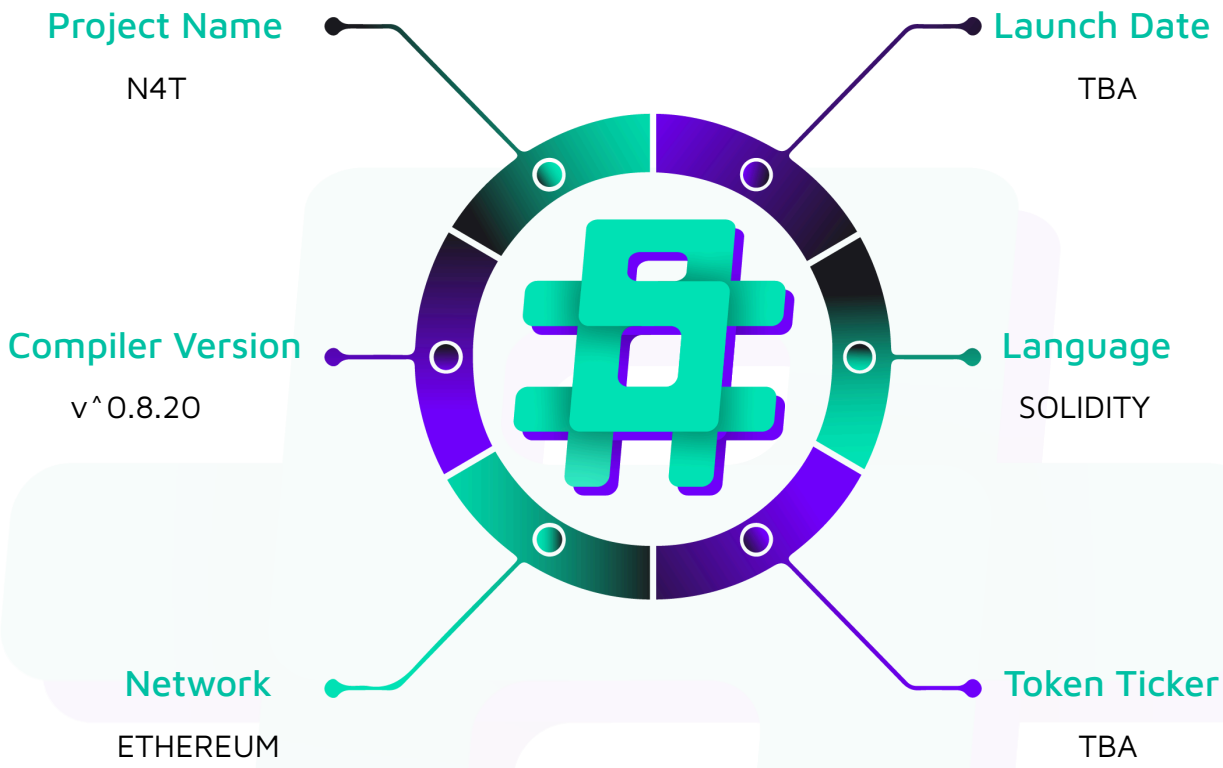
The N4T team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

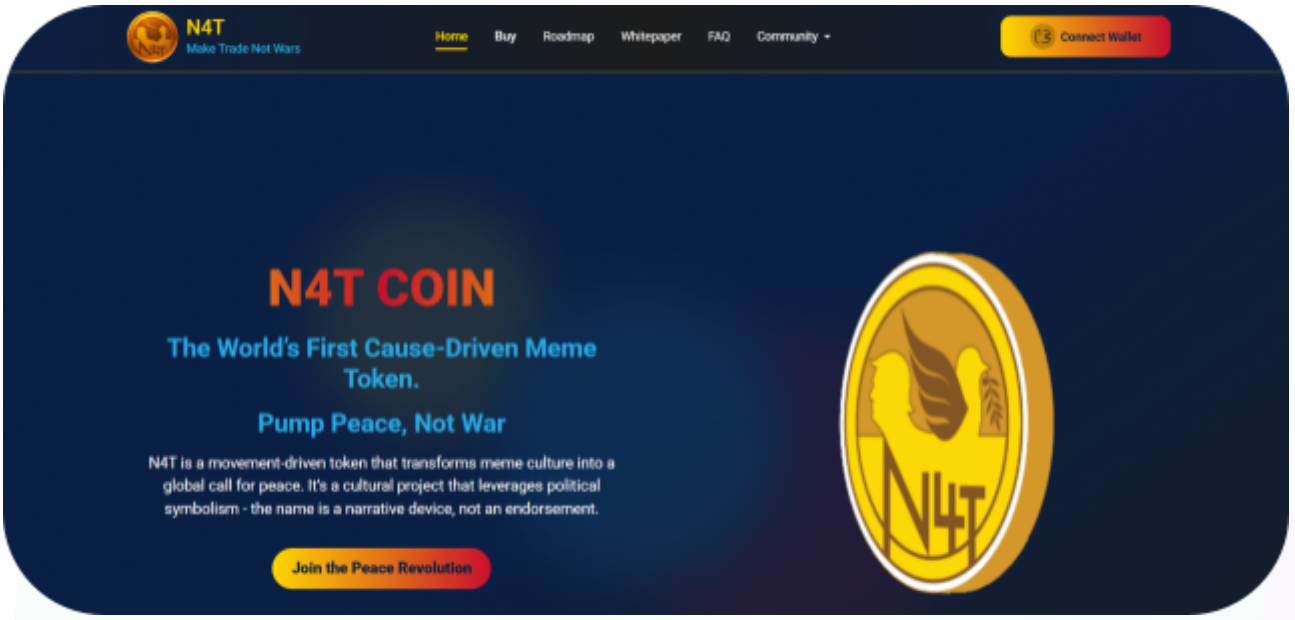
N4T is an innovative platform that integrates blockchain technology, digital economy, and community engagement to build a decentralized and transparent ecosystem. The project aims to connect users, developers, and organizations in a collaborative environment where digital assets, governance, and rewards are managed securely and autonomously. With a modern interface and a strong focus on usability, N4T seeks to democratize access to Web3 solutions, fostering new forms of interaction and participation within the decentralized economy.

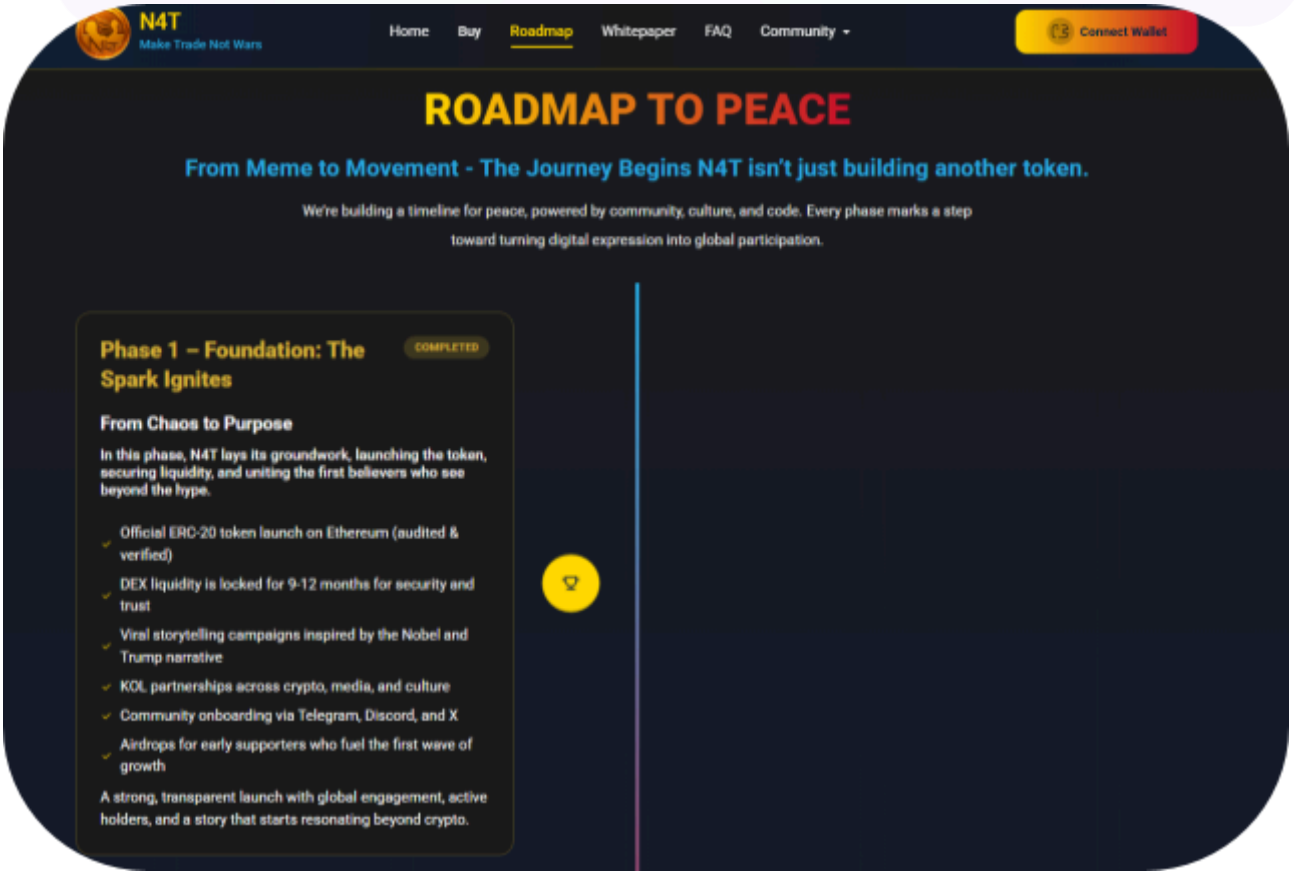
Project Name: N4T
Project Type: dApp
Compiler Version: ^0.8.20
Website: https://n4t.io/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the N4T 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	**N4T Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**September, 2025**
Contract 1	ReentrancyGuard.sol
Contract 2	ICO.sol
Contract 3	N4TToken.sol
Contract 4	OwnedUpgradeabilityProxy.so
Audited GitHub Commit Hash	fa4a6a6b8aba5f560c0efc1fc0fb499ae5bd91e8
Fix Review GitHub Commit Hash	94e4592ae20f855fbc66d3ebbdf5b77877ac8faa

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:

4 High severity vulnerabilities

5 Medium severity vulnerabilities

5 Low severity vulnerabilities

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
ICO.sol - Manages a multi-phase token sale (3 phases) with per-phase limits, prices and percentages. - Accepts ETH, USDT, USDC, calculates token amounts via oracles/placeholders and handles payments to a receiver. - Registers buyers for vesting schedules, supports claiming over time, and allows owner actions (pause/receiver update/send leftover). - Uses internal phase progression, vesting mappings and a reentrancy guard.	**Contract achieves this functionality.**
OwnedUpgradeabilityProxy.sol - Upgradeable proxy storing implementation, owner and maintenance flag in custom storage slots. - Owner can upgrade implementation and optionally call the new implementation (upgradeTo / upgradeToAndCall). - Fallback/receive delegatecalls to current implementation; maintenance mode restricts calls to proxy owner. - Uses low-level assembly for storage and delegatecall.	**Contract achieves this functionality.**
N4TToken.sol	**Contract achieves this**

- ERC20 token with Ownable-controlled minting, burnable and pausable features. - Implements transfer taxes (separate buy/sell and transfer rates) and forwards fees to a taxCollector. - Maintains wallet whitelists (tax-exempt) and blacklists (blocked transfers) and detects AMM buy/sell by checking if counterparty is a contract. - Owner can update tax rates, tax collector, and lists.	**functionality.**

Code Quality

This audit scope involves the smart contracts of the N4T 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 N4T 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] ICO.sol#initialize - Founders vesting (choice 5) sets 9 claims but only 7 tranche percentages, causing out-of-bounds access during claims

Description

Mismatch between `vestingInfoMapping[5].claims` and `percentageArray[5].length` will revert founder claims due to the array index being out of bounds.

Vulnerability Details

In `initialize()`, founders (choice 5) are configured with 9 claims, but `percentageArray[5]` has only 7 entries

```
setVestingCategory( currentTime + 165 minutes + 9 minutes, 3 minutes, 5, 9 ); // founders: claims = 9

percentageArray[5] = [ 800_000_000, 1_533_333_333, 1_533_333_333, 1_533_333_333, 1_533_333_333, 1_533_333_333, 1_533_333_333 ];
```

`vestingCalulations()` indexes tranche percentages by claim index `i`, assuming the array has at least `_claimCount` elements:

```
for (uint8 i = _userClaimCount; i < _claimCount; ++i) {

    if ((percentageArray[_choice]).length == 1) {

        amount += (_userTotalAmount * (percentageArray[_choice][0])) / 10_000_000_000;

    } else {

>>      amount += (_userTotalAmount * (percentageArray[_choice][i])) / 10_000_000_000;
}

}
```

When `_choice == 5` and `_claimCount == 8`, the loop tries to read `percentageArray[5][7]` which does not exist (length 7, valid indices 0–6), causing a panic and reverting the claim. Although the final claim path (`_claimCount == claims`) bypasses the loop, any intermediate claim at count 8 will consistently revert.

Impact

Denial of service for founders' claims: Founders cannot complete the 8th claim; the function reverts due to out-of-bounds array access.

Recommendation

A: `percentageArray[5]` contain 9 elements whose sum is `10_000_000_000` (100% in 1e-10 units) to match claims = 9.

B: Change `setVestingCategory(..., 5, 9)` to `setVestingCategory(..., 5, 7)` so that claimsmatches the 7-element array.

Status

Resolved

[H-02] ICO.sol#buyTokens - Not Validating msg.value When Token Type is Stable Coins

Description

`buyTokens()` is payable but does not validate `msg.value == 0` for non-ETH purchases (`_type 2=USDT`, `3=USDC`). If users attach ETH while buying with stablecoins, the transaction succeeds and the ETH remains stuck in the contract.

Vulnerability Details

The `buyTokens` function is declared payable to support ETH purchases but does not restrict ETH transfers when users buy tokens using stablecoins (`_type == 2` for `USDT`, `_type == 3` for `USDC`).

As a result, a user can accidentally attach msg.value while choosing a stablecoin purchase type. Since the function logic only processes `_usdtAmount` and transfers ERC20 tokens, any attached ETH remains in the contract. The missing validation of `msg.value == 0` for non-ETH paths is the root cause of this issue.

```
function buyTokens(uint8 _type, uint256 _usdtAmount) external payable {

    uint8 _phase = getCurrentPhase();

    // ...

    uint256 _buyAmount;

    IERC20Metadata _instance;

    if (_type == 1) {

        _buyAmount = msg.value;                          // ETH path

    } else {

        _buyAmount = _usdtAmount;                        // USDT/USDC path

        _instance = _type == 2 ? usdtInstance : usdcInstance;

        // allowance/balance checks, then transfer stable to receiver

                    TransferHelper.safeTransferFrom(address(_instance),   _msgSender(),
receiverAddress, _buyAmount);
```

```
    }

    // ... no check that msg.value == 0 on non-ETH path

}
```

Impact

Users who send ETH along with stablecoin purchases will have their ETH trapped in the ICO contract. These funds cannot be refunded or recovered through any existing function, resulting in permanent loss and potential legal or reputational exposure for the project.

Recommendation

Add explicit validation to enforce mutual exclusivity between ETH and stablecoin inputs, and implement a controlled rescue function for accidentally sent ETH.

Status

Resolved

[H-03] ICO.sol#cryptoValues - Hardcode Oracle Values Mis-Value USDT/USDC And ETH

Description

The `cryptoValues()` function returns static, hardcoded token-to-USD conversion rates instead of using live Chainlink oracle data. This defeats the purpose of the OracleWrapper references set in the contract and exposes the ICO to pricing inaccuracies or manipulation risks.

Vulnerability Details

The function is intended to fetch real-time exchange rates from Chainlink oracles (`ETHOracle`, `USDTOracle`, `USDCOracle`). However, the relevant lines replaced with fixed constants:

```
function cryptoValues(...) internal view returns (uint256, uint256) {

        //.. code

        if (_type == 1) {

            _amountToUSD = 10000 * 10 ** 8; //ETHOracle.latestAnswer();

            _typeDecimal = 10 ** 18;

        } else if (_type == 2) {

            _amountToUSD = 500 * 10 ** 8; //USDTOracle.latestAnswer();

            _typeDecimal = uint256(10 ** usdtInstance.decimals());

        } else {

            _amountToUSD = 500 * 10 ** 8; //USDCOracle.latestAnswer();

            _typeDecimal = uint256(10 ** usdcInstance.decimals());

        }

        return (_amountToUSD, _typeDecimal);

    }
```

This means token purchase calculations depend on outdated static values, not market prices. The root cause is the disabled use of latestAnswer() from the oracle contracts, making the conversion logic static and inaccurate over time.

Impact

Buyers can acquire vastly more tokens than intended (e.g., 1 USDT treated as $500), effectively draining the sale allocation at negligible cost and permanently damaging tokenomics and proceeds.

Recommendation

Replace hardcoded constants with live oracle queries from Chainlink feeds and handle zero or stale price cases safely.

Status

Resolved

[H-04] ICO.sol#tokensToBeClaimed - Claim Counter Overflow Due To Uint8 Causes Permanent Claim Denial

Description

The `tokensToBeClaimed()` function uses an 8-bit integer (`uint8`) to calculate the vesting claim counter. Because `_claimCount` is derived as `uint8((_time / _vestingInfo.claimGap) + 1)`, it will overflow after 255 intervals and wrap around to zero. This design flaw leads to a situation where, after sufficient time passes, beneficiaries can no longer claim their remaining vested tokens.

Vulnerability Details

The overflow occurs when the number of elapsed claim intervals exceeds `255`. At that point, `_claimCount` resets to zero, and the conditional check if `(_claimCount <= user.claims)` becomes true. The function then `returns (0, _claimCount)`, which causes `claimTokens()` to report "`Nothing to claim right now.`" From that moment on, no additional claims can be made, even though unclaimed tokens remain.

This effectively locks the user's remaining vested tokens indefinitely. For example, with a `claimGap` of three minutes, the overflow arises after roughly twelve and a half hours; with a one-month claim gap, the same bug appears after about twenty-one years.

```
uint256 _amountToTransfer = ((_tokenAmount * phaseInfo[_phaseNo].percentage) / 10000);

_registerUser(_tokenAmount - _amountToTransfer, _phaseNo, _msgSender());
```

Scenario:

A beneficiary in the vesting category 4 or 5 has a `claimGap` of `3` minutes and waits 13 hours before claiming. At that point, `_time / claimGap ≈ 260`, which overflows the `uint8` counter to `4`. The contract believes the user has already completed claims and blocks further withdrawals, locking the user's tokens indefinitely.

Impact

Once the overflow is triggered, beneficiaries can no longer claim any tokens. All remaining vested amounts become permanently inaccessible, resulting in a total loss for the affected users. Depending on the configuration, the issue may occur quickly or only after a long vesting period, but the outcome is the same—an irreversible denial of service for claimants.

Recommendation

Perform claim count calculations using a larger integer type, such as `uint256` and clamp the computed value to the vesting schedule before converting. This ensures that the claim counter can never overflow.

Status

Resolved

Medium

[M-01] OwnedUpgradeabilityProxy.sol#setMaintenance - Maintenance Mode Blocks Both User Calls and Upgrade Functionality

Description

When `maintenance` mode is enabled, it not only restricts normal user interactions but also prevents the `upgradeToAndCall()` function from executing. This may cause operational issues during maintenance periods where the owner intends to perform contract upgrades or initializations.

Vulnerability Details

The `setMaintenance()` function sets a global maintenance flag that is checked inside the `_fallback()` function. When maintenance is true, all delegate calls are blocked unless the caller is the proxy owner.

However, the issue arises because `upgradeToAndCall()` performs an internal call to `address(this).call(data)`. Since this internal call triggers the fallback, it re-enters `_fallback()` — where the `maintenance()` check reverts the transaction, preventing the upgrade or initialization from completing. This effectively locks the proxy from being upgraded or reinitialized while in maintenance mode, even by the authorized owner.

```
function setMaintenance(bool _maintenance) external onlyProxyOwner {

    bytes32 position = maintenancePosition;

    assembly {

        sstore(position, _maintenance)

    }

}

function _fallback() internal {

    if (maintenance()) {

        require(
```

```
            msg.sender == proxyOwner(),
            "OwnedUpgradeabilityProxy: FORBIDDEN"
        );
    }
    ...
}
```

Impact

Owner cannot use `upgradeToAndCall()` during maintenance, preventing critical upgrades or fixes when the system is paused.

Recommendation

Refine the maintenance logic to only block external (user) calls, not internal upgrade operations.

Status

Acknowledged

[M-02] ICO.sol#tokensToBeClaimed - User Can Claim Vesting Tranche Immediately At Start Time

Description

`_claimCount` is computed with `a + 1`, enabling users to claim a vesting tranche exactly at `startTime` (no `cliff`). This can unintentionally release tokens earlier than intended.

Vulnerability Details

In the `tokensToBeClaimed` function, vesting tranches are calculated based on elapsed time since `_vestingInfo.startTime`. The function currently computes `_claimCount` as `(_time / _vestingInfo.claimGap) + 1`, allowing users to claim a tranche even when `_time == 0`. Normally, vesting should enforce a cliff period where no tokens are claimable until at least one full `claimGap` has passed.

However, by including `+1`, users can call `claimTokens` at the exact `startTime` and receive a vesting tranche earlier than intended. The root cause is the premature increment of `_claimCount` before verifying that any vesting interval has elapsed.

```
function setMaintenance(bool _maintenance) external onlyProxyOwner {

    bytes32 position = maintenancePosition;

    assembly {

        sstore(position, _maintenance)

    }

}

uint32 _time = uint32(block.timestamp - (_vestingInfo.startTime));

/* Claim in Ever Month 30 days for main net and 1 minutes for the test */

_claimCount = uint8((_time / _vestingInfo.claimGap) + 1);  //@audit

uint8 claims = uint8(_vestingInfo.claims);

}
```

Impact

Users can unlock and withdraw one tranche of tokens immediately at `startTime`, effectively bypassing any intended cliff or delay in vesting. This results in premature liquidity exposure, inconsistent vesting schedules, and weakened token lock-up guarantees.

Recommendation

Compute `_claimCount` without the `+1` increment and enforce a minimum elapsed gap before any claim becomes available.

Status

Acknowledged

[M-03] N4TToken.sol#_determineFeeBP - User Can Bypass Tax Logic Using Contract Intermediaries

Description

The `_determineFeeBP()` function attempts to detect automated market maker (AMM) interactions by checking whether from or to addresses contain contract code (`code.length > 0`). However, this heuristic is unreliable and can be manipulated. Attackers or users can use intermediary contracts to spoof this condition, causing the token to apply the lower tax rate intended for normal wallet transfers.

Vulnerability Details

The current approach assumes that any interaction involving a contract address (`code.length > 0`) is a buy/sell on an exchange. In practice, this logic fails for several reasons:

- Many legitimate wallets (e.g., Gnosis Safe, proxy wallets, bridges) are smart contracts with code.length > 0, resulting in false positives.
- Attackers can easily deploy intermediary contracts that reroute transfers, toggling the isBuySell flag to achieve the more favorable (lower) tax rate.
- Since the tax rate is based solely on code.length, users can always choose the cheaper path by sending tokens through a contract wrapper.

The root cause is using code size as a proxy for transaction intent instead of checking explicit AMM pair addresses.

```
function _determineFeeBP(address from, address to) internal view returns (uint16) {

    bool isBuySell = from.code.length > 0 || to.code.length > 0;

    if (isBuySell) return buySellTax;

    return transferTax;

}
```

Impact

Attackers and savvy users can avoid higher buy/sell taxes by routing transfers through custom contracts. This directly reduces protocol fee revenue and breaks the tokenomics assumptions that rely on correct tax application.

Recommendation

Track and validate AMM pair addresses explicitly, rather than relying on code.length. Maintain a mapping of authorized liquidity pool addresses and check against it when determining tax type.

Status

Acknowledged

[M-04] OwnedUpgradeabilityProxy.sol#_upgradeTo - Missing Validation Allows Zero Or Non-Contract Implementation In Upgrade

Description

The `_upgradeTo(address newImplementation)` function does not verify that the supplied `newImplementation` address is valid or contains executable code. It only checks that the new implementation is not equal to the current one before updating the proxy's implementation slot. This omission allows the contract to be upgraded to an address with no code or even to the zero address.

Vulnerability Details

Because `_upgradeTo` directly calls `setImplementation(newImplementation)` without ensuring that `newImplementation` is a deployed contract, a single faulty or malicious upgrade transaction can disable the entire proxy system. If the `implementation` is set to `address(0)`, all proxied calls revert due to the standard `_fallback` check that requires a non-zero implementation. If the implementation points to an externally-owned account (EOA) or a contract with no code, delegatecalls may still execute successfully but perform no operations, causing unexpected or silent failures. In either case, users interacting through the proxy experience denial of service or inconsistent application behavior until another upgrade restores a valid implementation.

```
function _upgradeTo(address newImplementation) internal {
        address currentImplementation = implementation();
        require(
            currentImplementation != newImplementation,
            "OwnedUpgradeabilityProxy: INVALID"
        );
        setImplementation(newImplementation);
        emit Upgraded(newImplementation);
    }
```

Impact

Upgrading to a zero or non-contract address halts all proxy functionality. Users are unable to execute core logic, funds may be locked, and system downtime persists until a new upgrade transaction is issued. Even if later corrected, the event undermines trust and could disrupt protocol operations.

Recommendation

Add explicit validation checks to ensure that any new `implementation` is both non-zero and contains deployed bytecode. For stronger guarantees, consider enforcing interface compliance (e.g., proxiableUUID) depending on the upgrade pattern in use.

Status

Resolved

[M-05] ICO.sol#buyTokens - User Can Be Frontrun Into Next Phase Due To Missing Slippage And Phase Boundary Checks

Description

The `buyTokens()` function does not validate that the current phase and token price remain unchanged between transaction submission and execution. Because token pricing and vesting parameters are phase-dependent, a frontrunner or network delay can cause a purchase intended for one phase to be executed in the next phase, leading to an unexpected token rate and vesting category.

Vulnerability Details

The function retrieves the active phase at execution time using `getCurrentPhase()` and then proceeds to compute `_tokenAmount` via `calculateTokens()`. However, it lacks any mechanism to lock the user's purchase to the phase they expected when signing the transaction. This omission allows frontrunning or reordering attacks where a miner, bot, or external user intentionally delays or prioritizes transactions until a new phase becomes active. When this occurs, the buyer's funds are processed under the new phase's rules—potentially at a higher price and with a different vesting schedule—without user consent.

Impact

Buyers can suffer unexpected slippage in token allocation and incorrect vesting assignment if their transaction is executed near phase boundaries. Attackers or validators could intentionally manipulate transaction timing to exploit phase transitions, undermining user trust and fairness in the sale.

Recommendation

Introduce slippage and phase consistency checks so that transactions revert if executed under different pricing or phase conditions than intended. Require users to specify the expected phase or minimum token amount before purchase execution.

Status

Acknowledged

Low

[L-01] N4TToken.sol#setLowerMaxAllowedTax - Misleading Comment in setLowerMaxAllowedTax Function

Vulnerability Details

The function comment suggests that the `maxAllowedTax` has a default cap of `20%` which can only be reduced (not increased). Also, while declaring the variable `maxAllowedTax` comment says the default is `20%` and adjustable but downwards only However, there is no validation enforcing this upper bound in the code, making the comment misleading and the intended behaviour unclear.

Recommendation

Clarify and enforce the intended behavior. If `20%` is meant to be the permanent cap, add a validation.

Status

Resolved

[L-02] Token, N4TToken, ICO - Use ownable2Step

Vulnerability Details

Using `Ownable` provides no confirmation step for ownership transfer. If the current owner mistakenly transfers ownership to the wrong address or a non-functional contract, control over the contract can be permanently lost. The safer pattern `Ownable2Step` introduced in OpenZeppelin ensures that ownership transfer is a two-step process — the new owner must explicitly call acceptOwnership()to finalize the transfer. This prevents accidental or malicious ownership loss.

Recommendation

Use `Ownable2Step` instead of `Ownable` for safer ownership management.

Status

Acknowledged

[L-03] ICO.sol#updateReceiverAddress - Missing event emission in updateReceiverAddress()

Vulnerability Details

The admin setter updates `receiverAddress` without emitting an event, hindering off-chain monitoring and audit trails for a critical payout parameter.

Recommendation

Emit a dedicated event and include both the previous and the new address.

Status

Resolved

[L-04] N4TToken.sol#constructor - Dead Code

Vulnerability Details

The contract's constructor includes multiple commented-out parameters and assignments (`_owner`, `_taxCollector`, `_investor`, `_founderA`, `_founderB`, `_founderC`, `_marketing`, `_treasury`, `_liquidityPool`) that are never utilized. Additionally, several hardcoded addresses, such as `investor`, `founderA`, `founderB`, and `founderC` are declared and initialized but not referenced anywhere else in the codebase. These variables serve no functional purpose in the current deployment.

Recommendation

If the variables are not supposed to be used anywhere, consider removing them from the contract.

Status

Resolved

[L-05] ICO.sol#buyTokens - Buyer Can Receive Incorrect TGE And Vesting Allocation When Purchase Spans Multiple Phases

Description

The `buyTokens()` function in `ICO.sol` applies the TGE percentage and vesting schedule only from the last phase encountered during a purchase, even when the transaction spans multiple phases. As a result, when a purchase consumes remaining tokens from one phase and continues into the next, all tokens are treated as if they belong to the final phase. This misclassifies vesting rules and immediate token release amounts.

Vulnerability Details

In the `buyToken` function, `_tokenAmount` may include tokens sourced from multiple phases, but `_phaseNo` represents only the last phase reached. The contract then calculates the entire TGE portion using the last phase's percentage and registers the entire vesting remainder under the same phase category.

```
uint256 _amountToTransfer = ((_tokenAmount * phaseInfo[_phaseNo].percentage) / 10000);

_registerUser(_tokenAmount - _amountToTransfer, _phaseNo, _msgSender());
```

Scenario:

1. Phase 2 has a 30% TGE and Phase 3 has a 50% TGE.
2. Bob sends a large enough payment that buys 60% of the tokens from Phase 2 (now exhausted) and 40% from Phase 3.
3. Instead of applying 30% TGE to Phase 2 tokens and 50% TGE to Phase 3 tokens, the function treats all tokens as Phase 3 purchases.
4. Bob receives 50% of his total token allocation immediately, even for the Phase 2 portion, and all vesting is registered under Phase 3's faster schedule.

The root cause is that the contract aggregates `_tokenAmount` across phases but computes the TGE and vesting schedule using only the final `_phaseNo`.

Impact

Users may receive a higher or lower immediate token release than designed, leading to unbalanced token distribution.

Recommendation

Track and compute token allocations per phase within the purchase logic.

For each phase segment, apply the correct `phaseInfo[_phase].percentage` and register vesting separately under that phase's category.

Status

Acknowledged

Centralisation

The N4T project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the N4T 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