hashlock.

2024

Security Audit

MedXT (dApp)

September 2024 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.

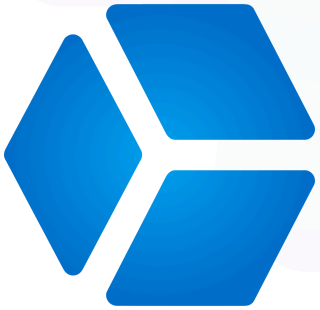
Executive Summary

The MediChainX team partnered with Hashlock to conduct a security audit of their CliffAndVesting.sol and MedXToken.sol 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.

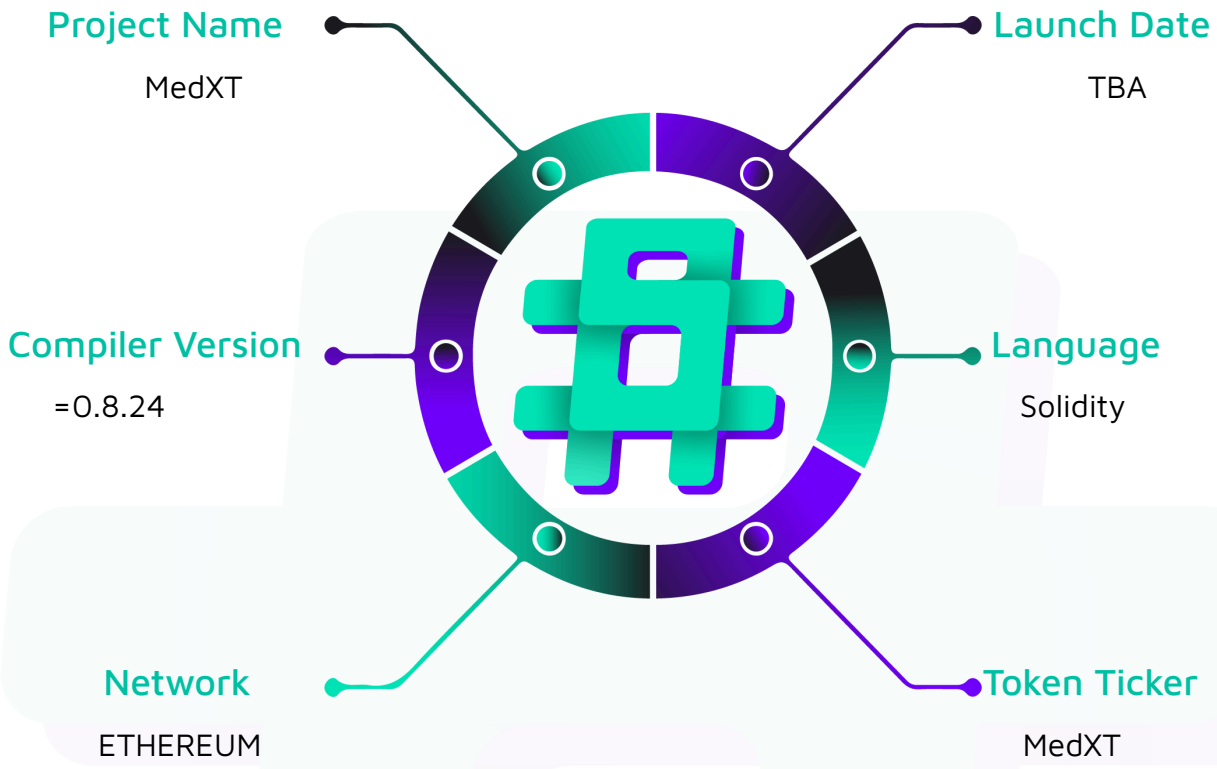
Project Context

MediChainX is a blockchain-based platform aimed at revolutionizing healthcare data management. It leverages Web 3.0 technologies to securely store and share electronic medical records (EMR), while ensuring HIPAA compliance and enhancing patient privacy. Patients and physicians gain seamless access to health data, fostering better collaboration and more informed decision-making. The platform focuses on privacy, data control, and compliance, allowing individuals to manage their healthcare information autonomously. MediChainX also introduces a healthcare wallet DApp and tracks pharmaceutical supply chains for transparency.

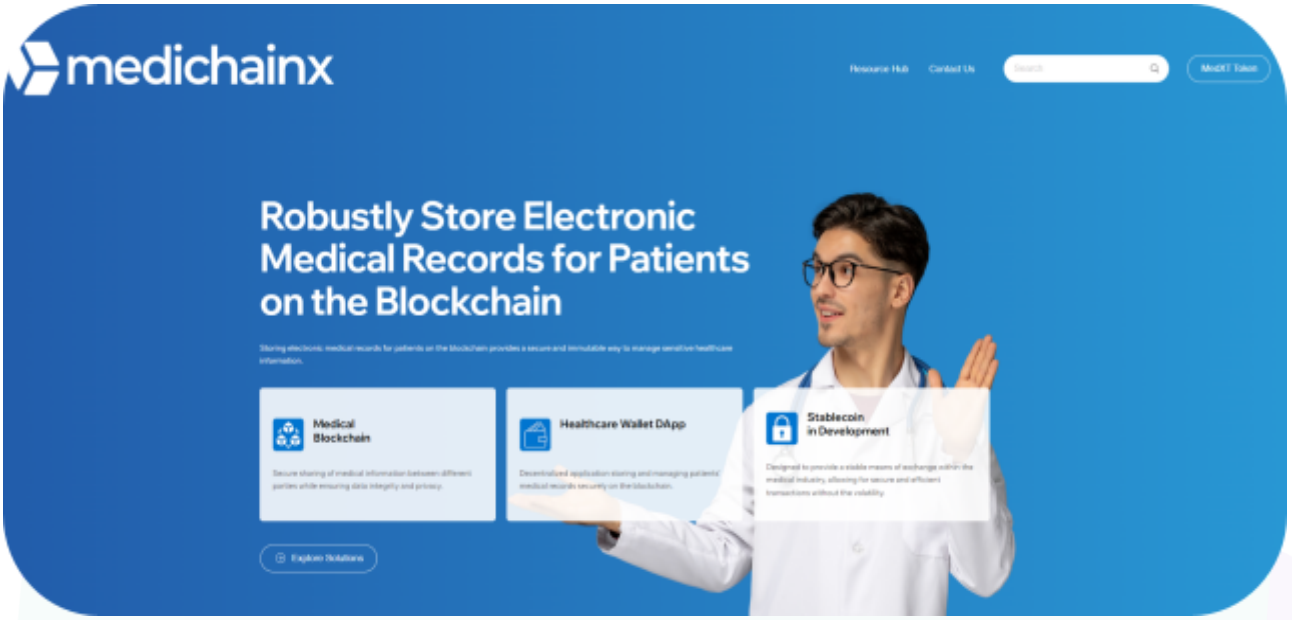
Project Name: MedXT
Compiler Version: =0.8.24
Website: www.medichainx.io
Logo:



Visualised Context:



Project Visuals:



Audit scope

We at Hashlock audited the solidity code within the MedXT project, the scope of works 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	MedXT Protocol Smart Contracts
Platform	Ethereum / Solidity
Audit Date	September, 2024
Contract 1	CliffAndVesting.sol
Contract 1 MD5 Hash	8182b463c2c3e61bb2ccf77dbdf103fd
Contract 2	MedXToken.sol
Contract 2 MD5 Hash	0395716eb4146fd2430acbb358494a95

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. All vulnerabilities initially identified have now been resolved and acknowledged.

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. General security overview is presented in the Standardised Checks section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

We initially identified some significant vulnerabilities that have since been addressed.

Hashlock found:

4 Medium severity vulnerabilities

4 Low severity vulnerabilities

3 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

CliffAndVesting.sol
- Allows users to:
  - Claim their vested tokens
- Allows owner to:
  - Reserve tokens for vesting
  - Start the vesting process
  - Claim tokens on behalf of multiple users
  - Claim tokens for a range of users
- Contract is used to:
  - Implement a token vesting schedule with a cliff period
  - Manage token distribution over time for multiple accounts
  - Provide a customizable vesting structure with initial release and periodic vesting
	Contract achieves this functionality.

MedXToken.sol
- Allows users to:
  - Transfer tokens (standard ERC20 functionality)
  - Burn tokens (if they are authorized burners)
- Allows owner to:
  - Update the admin address
  - Toggle fee collection
  - Update the fee receiver address
  - Manage whitelist and blacklist
  - Update the list of addresses subject to tax
  - Update the list of addresses authorized to
	Contract achieves this functionality.

burn tokens
- Contract is used to:
  - Implement an ERC20 token with additional features
  - Apply buy and sell fees on transfers
  - Manage whitelisted and blacklisted addresses	

Code Quality

This Audit scope involves the smart contracts of the MedXT 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 was required.

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

Audit Resources

We were given the MedXT project's 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 understanding the overall architecture of the protocol.

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

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

Audit Findings

Medium

[M-01] MedXToken#_updateBlacklist - No checks while blacklisting addresses can lead to critical addresses being blacklisted, breaking contract functionality

Description

The `updateBlacklist` function allows the owner or the admin to add or remove users from the blacklist. A blacklisted address can not receive or send MedX tokens. This function or the `_updateBlacklist` function has no checks implemented to make sure critical addresses can not be blacklisted.

Vulnerability Details

The `updateBlacklist` and `_updateBlacklist` functions shown below update the blacklist status of given addresses.

```
   function updateBlacklist(
        address[] calldata removeFromBlacklist,
        address[] calldata addToBlacklist
    ) external onlyOwnerOrAdmin {
        uint256 removeListLength = removeFromBlacklist.length;
                        for (uint256 i = 0; i < removeListLength; i++)
_updateBlacklist(removeFromBlacklist[i], false);
        uint256 addListLength = addToBlacklist.length;
                         for (uint256 i = 0; i < addListLength; i++)
_updateBlacklist(addToBlacklist[i], true);
    }
   function _updateBlacklist(address account, bool blacklist) private {
        if (blacklisted[account] == blacklist) return;
        blacklisted[account] = blacklist;
        emit BlacklistUpdated(account, blacklist);
    }
```

As observed in these functions, there are no checks implemented that would stop certain critical addresses from being blacklisted. A potential mistake leading to blacklisting of the UniSwap addresses or the `feeReceiver` address would break the contract functionality.

```
    function _update(address from, address to, uint256 value) internal override(ERC20)
{
        if (blacklisted[from]) revert Blacklisted(from);
        if (blacklisted[to]) revert Blacklisted(to);
        //rest of the function
```

Due to the code snippet shown below, if these addresses would be blacklisted, any transfer calls to these addresses would revert.

Impact

It would not be possible to do any token transfer that requires a tax or any token transfer that interacts with UniSwap such as buying or selling.

Recommendation

Implement checks in the functions to block certain critical addresses from being blacklisted. An example is shown below:

```
    function _updateBlacklist(address account, bool blacklist) private {
        if (account == feeReceiver || account == uniV2Router) revert InvalidInput();
        if (blacklisted[account] == blacklist) return;
        blacklisted[account] = blacklist;
        emit BlacklistUpdated(account, blacklist);
    }
```

Status

Resolved

[M-02] MedXToken#_updateWhitelist - Addresses that are added to the taxed list can be whitelisted resulting in these addresses bypassing taxes

Description

The `updateWhitelist` function allows the owner or the admin to add or remove addresses from the whitelist. Being in the whitelist means these addresses can send or receive tokens without any taxes. This function or the `_updateWhitelist` function has no checks to make sure taxed addresses can not be whitelisted.

Vulnerability Details

The `updateWhitelist` and the `_updateWhitelist` functions shown below update the whitelist status of given addresses.

```
   function updateWhitelist(
        address[] calldata removeFromWhitelist,
        address[] calldata addToWhitelist
    ) external onlyOwnerOrAdmin {
        uint256 removeListLength = removeFromWhitelist.length;
                        for (uint256 i = 0; i < removeListLength; i++)
_updateWhitelist(removeFromWhitelist[i], false);
        uint256 addListLength = addToWhitelist.length;
                         for (uint256 i = 0; i < addListLength; i++)
_updateWhitelist(addToWhitelist[i], true);
    }

    function _updateWhitelist(address account, bool whitelist) private {
        if (whitelisted[account] == whitelist) return;
        whitelisted[account] = whitelist;
        emit WhitelistUpdated(account, whitelist);
    }
```

As observed in these functions, there are no checks implemented to make sure that the address being whitelisted is not a taxed address. Taking a look at the `_update` function below.

```
function _update(address from, address to, uint256 value) internal override(ERC20) {
        if (blacklisted[from]) revert Blacklisted(from);
        if (blacklisted[to]) revert Blacklisted(to);
              if (!feeEnabled || whitelisted[from] || whitelisted[to] || from == address(this) || to == address(this)) {
            return super._update(from, to, value);
        }
        if (applyTax[to]) _updateWithSellFee(from, to, value);
        else if (applyTax[from]) _updateWithBuyFee(from, to, value);
        else super._update(from, to, value);
    }
```

It is observed that a potential mistake whitelisting a taxed address (such as UniSwap pairs) would lead to these addresses avoiding tax.

Impact

Addresses that should be taxed on token transfers such as DEX pairs would not be taxed if they were accidentally added to the whitelist.

Recommendation

Implement a check in your functions to prevent taxed addresses from being added to the whitelist. An example is shown below.

```
   function _updateWhitelist(address account, bool whitelist) private {
        if (applyTax[account] == true && whitelist == true) revert InvalidInputs();
        if (whitelisted[account] == whitelist) return;
        whitelisted[account] = whitelist;
        emit WhitelistUpdated(account, whitelist);
    }
```

Status

Resolved

[M-03] MedXToken - Centralization risk for privileged functions

Description

Contracts with privileged functions need the owner to be trusted not to perform malicious updates. This may also cause a single-point failure.

Vulnerability Details

The `updateBlacklist` function allows the admin or the owner to add users to the blacklist. Addresses in the blacklist will lose the ability to transfer and receive tokens.

```
   function updateBlacklist(
        address[] calldata removeFromBlacklist,
        address[] calldata addToBlacklist
    ) external onlyOwnerOrAdmin {
        uint256 removeListLength = removeFromBlacklist.length;
                        for (uint256 i = 0; i < removeListLength; i++)
_updateBlacklist(removeFromBlacklist[i], false);
        uint256 addListLength = addToBlacklist.length;
                         for (uint256 i = 0; i < addListLength; i++)
_updateBlacklist(addToBlacklist[i], true);
    }
    function _update(address from, address to, uint256 value) internal override(ERC20)
{
        if (blacklisted[from]) revert Blacklisted(from);
        if (blacklisted[to]) revert Blacklisted(to);
        //rest of the function
```

The `updateTaxedAddresses` function allows the admin or owner to apply tax to all token transfers of or to an address. This function is meant to be used for DEX pair addresses to implement a tax on token trades. However, no restrictions in this function means that this function can be used maliciously, either on accident or on purpose.

```
   function updateTaxedAddresses(
        address[] calldata removeFromTaxedList,
        address[] calldata addToTaxedList
    ) external onlyOwnerOrAdmin {
```

```
        uint256 removeListLength = removeFromTaxedList.length;

                        for (uint256 i = 0; i < removeListLength; i++)
_updateTaxedList(removeFromTaxedList[i], false);

        uint256 addListLength = addToTaxedList.length;

                        for (uint256 i = 0; i < addListLength; i++)
_updateTaxedList(addToTaxedList[i], true);

    }
```

Impact

Privileged roles have the ability to cause any user to be taxed on normal token transfers or blacklist any user to deny their ability to transfer or receive tokens. This introduces a single point of failure. A malicious or compromised privileged role can negatively impact users.

Recommendation

To mitigate centralization risks associated with privileged functions, consider implementing a multi-signature or decentralised governance mechanism. Instead of relying solely on a single owner/administrator, distribute control and decision-making authority among multiple parties or stakeholders. This approach enhances security, reduces the risk of malicious actions by a single entity, and prevents single points of failure.

Note

According to the MedXT team, implementation is done according to the context and requirements from the customer.

Status

Acknowledged

[M-04] MedXToken - It is possible for users to launch new pairs to avoid taxes applied to DEX trading

Description

MedXToken has a 3% fee built in on DEX trades (i.e buying or selling on UniSwapV2 pairs.) The addresses that the tax will apply to are added manually, meaning that the protocol might forget to add pairs to be taxed, users would prefer to trade on the pairs that are not yet taxed or users would launch more and more pairs, turning it into a race.

Vulnerability Details

The `constructor` in the contract creates two new pairs for the token and adds them to the `applyTax` mapping.

```
   constructor(
        address _owner,
        address payable _feeReceiver,
        IUniswapV2Router02 _uniV2Router,
        address usdt
    ) ERC20("MedXT", "$MedXT") Ownable(_owner) {
        //rest of the constructor
        address wethUniV2Pair = factory.createPair(self, weth);
        address usdtUniV2Pair = factory.createPair(self, usdt);
        _updateTaxedList(wethUniV2Pair, true);
        _updateTaxedList(usdtUniV2Pair, true);
        _approve(self, address(_uniV2Router), type(uint256).max);
    }
```

When the token is launched only these two pairs will have tax applied to them. Take a look at the `_update` function below

```
   function _update(address from, address to, uint256 value) internal override(ERC20)
{
        if (blacklisted[from]) revert Blacklisted(from);
        if (blacklisted[to]) revert Blacklisted(to);
```

```
            if (!feeEnabled || whitelisted[from] || whitelisted[to] || from ==
address(this) || to == address(this)) {
            return super._update(from, to, value);
        }
        if (applyTax[to]) _updateWithSellFee(from, to, value);
        else if (applyTax[from]) _updateWithBuyFee(from, to, value);
        else super._update(from, to, value);
    }
```

It is observed that the tax is applied to only the addresses in the `applyTax` mapping. There are no dynamic checks to figure out if the to or from addresses are DEX pairs which will lead to users launching new pairs to avoid taxes. For example, the MedX-USDT pair is taxed but users can launch a MedX-USDC pair to do the same trades without taxes. Since the addresses are added manually by the owner or the admin with the `updateTaxedAddresses` function any trades done before the pair is taxed will cause the protocol to lose funds and the longer it takes to update the taxed addresses the more funds the protocol will lose.

Impact

The protocol will lose out on taxes they intend to get from DEX trades.

Recommendation

Implement a system that dynamically checks if the to or from address in the `_update` function is a DEX pair. This can be done by adding a new `isDEXPair` function that can look for common patterns in DEX pairs. Do keep in mind that this solution will be gas expensive.

Additionally, it would be better to have an off-chain bot that constantly tracks for any pairs in any DEX, including MedX, and adds it to the taxed list automatically. Keep in mind that these bots would need to have admin or owner privileges.

Note

Taxes will only be applied to the pairs explicitly defined in the constructor. This vulnerability, which allows users to create untaxed trading pairs and potentially scale the issue, was presented to the MedXT team. The MedXT team acknowledged that while

this could result in missed tax collection, implementing a solution that checks each transaction dynamically would significantly increase gas costs. As a result, the current strategy was approved, accepting the missed tax collection.

The MedXT team also noted that the team taxes uniswap v2, but in uniswap v2 the users cannot create more than one pair, thus a malicious user is not able to create multiple pairs. As well as privileged roles can cause any user to be taxed at 3% or blacklist any user to deny their ability to transfer or receive tokens. This introduces a single point of failure. A malicious or compromised privileged role can negatively impact users very easily

Status

Acknowledged

Low

[L-01] CliffAndVesting - Missing checks for address(0) in constructor

Description

The constructors lack `address(0)` checks for the parameters shown below. Implementing `address(0)` checks can prevent accidental inputs or cause an earlier revert.

```
_token // @CliffAndVesting.sol
```

Recommendation

Implement `address(0)` checks in the constructors.

Status

Resolved

[L-02] MedXToken#_updateWithSellFee, _updateWithBuyFee - Low-value transfers will not pay taxes

Description

Due to how Solidity handles divisions, any transfer value that is less than 34 will avoid any tax as the fee will be 0. However, due to gas costs, this is a very unlikely scenario

```
   function _updateWithSellFee(address from, address to, uint256 value) private {
        uint256 fee = (value * SELL_FEE_PERCENT) / 100;
        value = value.unsafeSub(fee);
        _collectFee(from, to, from, fee);
        super._update(from, to, value);
    }
      function _updateWithBuyFee(address from, address to, uint256 value) private
nonReentrant {
        uint256 tokenFee = (value * BUY_FEE_PERCENT) / 100;
```

```
        value = value.unsafeSub(tokenFee);
        address self = address(this);
        super._update(from, self, tokenFee);
        (bool swapSucceed, uint256 ethFee) = _trySwapFeeToEth(tokenFee);
             if (swapSucceed) emit FeeEthCollected(from, to, feeReceiver, tokenFee,
ethFee);
        else _collectFee(from, to, self, tokenFee);
        super._update(from, to, value);
    }
```

Recommendation

Consider implementing a minimum fee amount of 1 or do not allow transfers of less than 34 tokens.

Status

Acknowledged

Note

The MedXT team advised adding a suggested check would increase gas on all the transfers. As the loss is really low 0.000000000000000001 MEDXT if transfer < 0.000000000000000034 MEDXT and considering high gas prices (Ethereum) they considered that as not relevant to adjust.

[L-03] MedXToken#updateAdmin - Setting a new admin should be a two-step process

Description

Updating an admin should be a two-step process to reduce the risk of setting an admin to the wrong address. A two-step process such as Ownable2Step by OpenZeppelin but for the admin role that requires the new address to accept the role will add another layer of security to the critical admin role

```
      function updateAdmin(address newAdmin) external onlyOwnerOrAdmin returns (bool
changed) {
        return _setAdmin(newAdmin);
    }
    function _setAdmin(address newAdmin) private returns (bool changed) {
        if (admin == newAdmin) return false;
        emit AdminUpdated(admin, newAdmin);
        admin = newAdmin;
        return true;
    }
```

Recommendation

Make the `updateAdmin` function a two-step process that requires the new address to accept the role.

Note

The MedXT team inferred the owner already uses a two-step process on the contract, and since the owner can set the admin, the admin role will never be lost.

Status

Acknowledged

[L-04] Contracts - Floating pragma

Description

Floating pragmas are used without a specific version number, allowing the contract to be compiled with the latest available compiler version. This can lead to various compatibility and stability issues.

Recommendation

Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs for the compiler version that is chosen.

Status

Resolved

Gas

[G-01] CliffAndVesting#vestingName - Do not cache constants to save gas

Description

This practice is unnecessary and can actually increase gas consumption due to the additional storage operations for the local variables.

```
function vestingName() external view returns (string memory _vestingName) {
     bytes32 _vestingId = vestingId;
     _vestingName = string(new bytes(uint256(_vestingId >> 0xf8)));
     assembly {
         mstore(add(_vestingName, 31), _vestingId)
     }
 }
```

Recommendation

Refrain from caching constant variables into local variables.

Status

Acknowledged

[G-02] CliffAndVesting#MAX_RESERVE_AMOUNT - Do not calculate constants

Description

Due to how constant variables are implemented (replacements at compile-time), an expression assigned to a constant variable is recomputed each time that the variable is used, which wastes some gas.

```
uint256 private constant MAX_RESERVE_AMOUNT = type(uint256).max / 1e18;
```

Recommendation

Precompute and assign literal values to constant variables. Avoid defining constants with expressions or calculations.

Status

Resolved

[G-03] MedXToken - State variables that are used multiple times in a function should be cached in stack variables

Description

When performing multiple operations on a state variable in a function, it is recommended to cache it first. Either multiple reads or multiple writes to a state variable can save gas by caching it on the stack. There are 3 instances of this issue

```
line 173: emit FeeTokenCollected(transferFrom, transferTo, feeReceiver, feeValue);
//State variable feeReceiver is used also on line 172.

line 222: emit FeeReceiverUpdated(feeReceiver, newFeeReceiver);  //State      variable
feeReceiver is used also on line 220.

252: if (admin == newAdmin) return false; //State variable admin is used also on line
253.
```

Recommendation

Cache state variables in stack or local memory variables within functions when they are used multiple times.

Status

Resolved

Centralisation

The MedXT 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 Hashlocks analysis, the MedXT project seems to have a sound and well tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented 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 are 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 whitebox 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, 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 contracts 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 are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree 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 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