Hashlock.

ArmSwap

(Protocol)

SMART CONTRACT

Security Audit

Platform
ETH

hashlock.com.au
APRIL 2024

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 ArmSwap partnered with Hashlock to conduct a security audit of their Armswap-token and armswap-evm-contract-main. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts were secure.

Project Context

ARMSwap is a decentralised application (DApp) that facilitates cross-chain token swaps and bridge crypto assets across different blockchains. Users can engage in native-native swapping, native-ERC20 token swapping, native-WrappedNative Bridging, and ERC20-ERC20 Swapping.

Project Name: ArmSwap
Compiler Version: ^0.8.19 (ARMswap Token)
Compiler Version: ^0.8.24 (ArmSwap)
Website: https://www.armswap.com/
Logo:

Visualised Context:

Project Name	ArmSwap
Launch Date	June 2024
Compiler Version	v^0.8.24 / ^0.8.19
Language	Solidity
Network	ETHEREUM
Token Ticker	ARMSP

Iconography:

Project Visuals:

Pioneering the Future of Blockchain Interoperability

Enabling Smooth Cross Chain Connectivity for a Decentralized Future

Audit scope

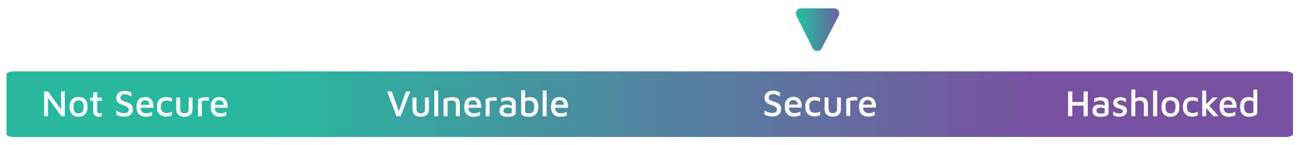
We at Hashlock audited the solidity code within the Armswap-token and armswap-evm-contract-main, the scope of works included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**armswap-evm-contract-main**
Platform	**Ethereum / Solidity**
Audit Date	**April 2024**
ArmswapV1Router.sol	4203613109e1c33b860f7cf634300fbf
ArmswapV1ERC20	ba3a536e99c2b1229a41d32d09ed4da0
ArmswapV1ERC20Deployer.sol	e1b87a6706b3e84b9ed8b5578f0a4057
RouterConfig.sol	e621685c126da0bde1ad20a66deae0b2
ArmCallExecutor.sol	1f89c20a12152fed20b3bcb8e5794d05
ArmswapV1RouterSecurity.sol	93c93a0fccfc25d07c42475c743b3c1d

Description	**armswap-token**
Platform	**Ethereum / Solidity**
Audit Date	**April 2024**
ARMswap.sol	1763307074eb15ebf4e2748b31459fff
BlackList.sol	6f09071b2472ace95c39bfae19d9896d
TransferControl.sol	be6966bde491355aa9b3fe88a6dd6c34

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



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.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

1 High severity vulnerability

10 Medium severity vulnerabilities

3 Low severity vulnerabilities

2 Gas Saving

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 Behaviours

Claimed Behaviour	Actual Behaviour
ARMswap.sol - Allows users to: - Transfer - Approve - Batch Transfer - Burn Token - Allows admins to: - Pause / Unpause the transfer - Add / Remove blacklist address - Enable token transfer - Add / Remove user token transfer capability	**Contract achieves this functionality with some issues**
BlackList.sol - Periphery contract used to: - Manage the blacklist user	**Contract achieves this functionality.**
TransferControl.sol - Periphery contract used to: - Manage transfer allowed	**Contract achieves this functionality with some issues**
ArmswapV7Router.sol - Allows users to: - Create Pool - Swap - Allows admins to: - Set Router Security - Change Vault - Extract Fees	**Contract achieves this functionality with some issues**

ArmswapV6ERC20Deployer.sol - Periphery contract used to: - Deploy new pair	**Contract achieves this functionality.**
ArmswapV6ERC20.sol - Periphery contract used to: - Deposit - Withdraw - Mint - Burn	**Contract does not achieves this functionality.**
RouterConfig.sol - Periphery contract used to: - Set Chain Config - Set Fee Config - Set Token Config - Set Swap Config - Set Custom Config	**Contract achieves this functionality.**
ArmCallExecutor.sol - Periphery contract used to: - Add Caller - Remove Caller - Execute	**Contract achieves this functionality.**

Code Quality

This Audit scope involves the smart contracts of the ArmSwap, 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 smart contract codes for Armswap-token and armswap-evm-contract-main in the form of a zip file.

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

High

[H-01] ARMswap#approve - Approve() can be frontrun

Description

Approve() are subject to front-run attack because the approve method overwrites the current allowance regardless of whether the spender already used it or not. In case the spender spends the amount, the approve() will approve a new amount.

Vulnerability Details

Approve() is subject to a known front-running attack.

```
function approve(address spender,uint256 value)public virtual override
whenNotPaused whenNotBlackListed(_msgSender(), spender) returns (bool) {

    address owner = _msgSender();

    _approve(owner, spender, value);

}
```

Attack Scenario

Here is a possible attack scenario:

1. Alice allows Bob to transfer N of Alice's tokens ($N > 0$) by calling the approve method on a Token smart contract, passing the Bob's address and N as the method arguments
2. After some time, Alice decides to change from N to M ($M > 0$) the number of Alice's tokens Bob is allowed to transfer, so she calls the approve method again, this time passing the Bob's address and M as the method arguments
3. Bob notices the Alice's second transaction before it was mined and quickly sends another transaction that calls the transferFrom method to transfer N Alice's tokens somewhere

4. If the Bob's transaction will be executed before the Alice's transaction, then Bob will successfully transfer N Alice's tokens and will gain an ability to transfer another M tokens
5. Before Alice noticed that something went wrong, Bob calls the transferFrom method again, this time to transfer M Alice's tokens.

So, an Alice's attempt to change the Bob's allowance from N to M ($N > 0$ and $M > 0$) made it possible for Bob to transfer $N + M$ of Alice's tokens, while Alice never wanted to allow so many of her tokens to be transferred by Bob.

Impact:

Possible for a user to over spend their allowance.

Recommendation

Consider using SafeERC20 with safeIncreaseAllowance() and safeDecreaseAllowance()

Note: These two functions have been deprecated by openZeppelin so use them with care.

Status

Resolved

Medium

[M-01] ArmswapV7Router#createERC20Pool - Unhandled return value of transferFrom could lead to potential fund loss

Description

ERC20 implementations are not always consistent. Certain implementations of transferFrom functions might return 'false' upon failure instead of reverting. To ensure safety, it's advisable to always check the result of transferFrom and transfer

Vulnerability Details

The createERC20Pool function does not check the return value from the transferFrom function. The contract might lose the funds if the underlying token does not revert the transaction.

```
function createERC20Pool(string memory _name,string memory _symbol, uint8
_decimals , address _underlying ,uint _liquidity,uint _dstID)external  {

   address newPool =
deployer.deployNewPair(_name,_symbol,_decimals,_underlying, admin,
wNATIVE,address(this));

   //does not check the result

   IERC20(_underlying).transferFrom(msg.sender,address(this),_liquidity);

   IERC20(_underlying).approve(newPool,_liquidity);

   IERC20(newPool).deposit(_liquidity,msg.sender);

   emit poolCreated(address(newPool), _dstID,_name, _symbol, _decimals);

}
```

Impact

Potentially losing funds due to unchecked return value from the transferFrom function

Recommendation

Verify the return value of transferFrom operations.

Status

Resolved

[M-02] ArmswapV6ERC20#_withdraw - Use of transfer() can result in revert

Description

The `transfer()` function uses a fixed amount of gas, which restricts protocols from interacting with other contracts that require more gas to process transactions, potentially leading to transaction failures.

Vulnerability Details

The `_withdraw` function uses `transfer()`, which could cause the transaction to fail if the receiver smart contract consumes more than 2300 gas units.

```
function _withdraw(address from, uint amount, address to) internal returns
(uint) {

     require(!underlyingIsMinted);

     require(underlying != address(0) && underlying != address(this));

     _burn(from, amount);

     if (underlying==wNative){

            IERC20(underlying).withdraw(amount);

            payable(to).transfer(amount);

    }else {

            IERC20(underlying).safeTransfer(to, amount);

    }

    return amount;

}
```

Impact

The `transfer()` function's use of a fixed amount of gas can lead to a revert.

Recommendation

Use call() instead of transfer().

Status

Resolved

[M-03] ARMswap#transfer & transferFrom - whenNotPaused is not implemented in Transfer and TransferFrom

Description

Missing the whenNotPaused modifier in Transfer() and TransferFrom() allows users to perform the actions even when the admin has paused the contract.

Vulnerability Details

As shown in the code snippet below, the transfer() does not implement the whenNotPaused modifier. This allows the user to transfer tokens even when the token has been paused by the owner.

```
function transfer(address to,uint256 value)public virtual override
requireTokenTransferEnabled(_msgSender()) whenNotBlackListed(_msgSender(),
to) returns (bool) {

     address sender = _msgSender();

     _transfer(sender, to, value);

     return true;

}
```

The following is another example that demonstrates the absence of the whenNotPaused modifier in the transferFrom().

```
function transferFrom(address from,address to,uint256 value)public virtual
override requireTokenTransferEnabled(_msgSender())
whenNotBlackListed(from, to) requireNotBlackListed(_msgSender()) returns
(bool) {

     address spender = _msgSender();

     _spendAllowance(from, spender, value);

      _transfer(from, to, value);

}
```

Impact

Users can still execute the transfer() and transferFrom() functions even when the admin has paused the contract.

Recommendation

Implement whenNotPaused in transfer() and transferFrom()

Status

Resolved

[M-04] ARMswap#enableTokenTransfer - Inability to disable the token transfer after activation

Description

The contract does not have the ability to disable token transfers again after activating them via the enableTokenTransfer().

Vulnerability Details

After the owner calls the enableTokenTransfer() from ARMSwap.sol, the contract invokes _enableTokenTransfer() from TransferControl.sol to set the _isTokenTransferEnabled value to true, allowing users to perform transfers. However, the contract lacks the ability to set the _isTokenTransferEnabled value to false, which prevents the owner from disabling token transfers after enabling it.

```
//From ARMSwap.sol

function enableTokenTransfer() external onlyOwner {

     _enableTokenTransfer();

}

//From TransferControl.sol

function _enableTokenTransfer() internal virtual {

     _isTokenTransferEnabled = true;

     emit TokenTransferEnabled();
}
```

Impact

The admin does not have the ability to disable token transfers from the user.

Recommendation

Implement a `disableTokenTransfer()` to set the `_isTokenTransferEnabled` to `false`.

Note

The absence of a disableTokenTransfer() function is intentional and reflects the contract's design choice. By omitting this feature, the contract emphasizes a commitment to continuous token transfer capabilities once enabled.

Status

Acknowledged

[M-05] ARMswapV6ERC20#withdrawVault - Centralization Risk - Enable to withdraw user funds

Description

The `withdrawVault()` function allows the vault owner to pass the `from` address to _withdraw(), enabling the owner to retrieve any user funds without their knowledge or approval.

Vulnerability Details

The Vault owner can call `withdrawVault()` to retrieve any user funds from the `from` address

```
function withdrawVault(address from, uint amount, address to) external
onlyVault returns (uint) {

     //from address not hard coded as msg.sender. Allow vault owner to
retrieve the funds

     return _withdraw(from, amount, to);
}
```

Impact

The vault owner can retrieve any user's funds without their approval.

Recommendation

- Make the vault owner multi-sig.

Status

Resolved

[M-06] ArmswapV6ERC20#setVault - Vault address can't be changed after setting it to wrong address

Description

Vault address can't be changed after setting it to wrong address

Vulnerability Details

When deploying a new pair using the "deployNewPair" function there isn't any kind of check on the parameters so if mistakenly the vault address has been set to "address(0)" then there is no way to change it. If owner wants to change the vault address he can't do it as there is a "onlyVault" modifier on the "setVault" function and if we see the "onlyVault" modifier which is as follow

```
modifier onlyVault() {

        require(msg.sender == vault, "ArmswapV6ERC20: FORBIDDEN");

        _;

    }
```

So this means that only the vault can change the address but if vault is set to address(0) he won't be able to do it

Impact

Unable to change vault address

Recommendation

Allow the owner of the vault to change the address

Note:

The vault address is indeed set when the **deployNewPair** function is called. The admin address is passed as `Vault` from the **ArmswapV7Router**, which is initialized in the constructor of the contract. We have a require condition in place that checks whether the admin address is not the zero address.

Given this setup, the admin address passed to the **ArmswapV7Router** constructor is already validated to ensure it's not the zero address. Therefore, there's no need for additional checks on the parameters within the **deployNewPair** function.

Additionally, regarding the ability to change the vault address, if the vault is mistakenly set to **address(0)**, it indeed presents a problem. However, this scenario is prevented by the initial validation of the admin address. The owner of the contract, who has control over the admin address, can ensure that the vault address is correctly set during deployment.

Therefore, the risk of setting the vault address to **address(0)** is mitigated by the validation of the admin address, and there's no need for further checks within the **deployNewPair** function.

Status

Acknowledged

[M-07] ArmswapV6ERC20#_mint - No cap on total supply

Description

There isn't any type of check on the total supply of the token

Vulnerability Details

When depositing the amount the "_mint" function calls and if we see this function then there isn't any type of supply cap on the amount of token that will be minted so that the overall supply can inflate beyond bound and in the worst case scenario it can overflow.

I understand that in the "_withdraw" function the tokens are being burned but there should also be a max cap on the amount of tokens being minted

```
function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");


        _totalSupply += amount;
        balanceOf[account] += amount;
        emit Transfer(address(0), account, amount);
    }
```

Impact

Overall supply can inflate..

Recommendation

Set a max supply cap on the amount of tokens that can be minted

Status

Resolved

[M-08] ArmswapV6ERC20#_mint - RemoveSupportedCaller function can be Dos'ed

Description

RemoveSupportCaller function can be Dos'ed due to storing support callers in an array

Vulnerability Details

If we see abstract contract "RoleControl" in "ArmswapV7RouterSecurity.sol" file there is an array in that contract which is

```
address[] public supportedCallers;
```

Now if we see "addSupportedCaller" function then every new supported caller is being added in the array and if we see "RemoveSupportedCaller" function then we can see that first this function is making sure that the "caller" address being removed is actually a caller and then it is removing it from mapping by actually setting it to "false" and after that it is looping a condition till the length of the "supportedCallers" array. Now the issue is that if the length of the array is too Big then this function can run out of gas cause this function is looping through every "caller" in the array and checking the condition on it. Let's say if the desired caller is in the end of array then looping through the whole array will cost alot of gas and will probably make this function run out of gas.

```
function removeSupportedCaller(address caller) external onlyAdmin {

       require(isSupportedCaller[caller]);

       isSupportedCaller[caller] = false;

       uint256 length = supportedCallers.length;

       for (uint256 i = 0; i < length; i++) {

           if (supportedCallers[i] == caller) {

               supportedCallers[i] = supportedCallers[length - 1];

               supportedCallers.pop();

               return;

           }
```

```
        }
    }
```

Impact

Function can be Dos’ed permanently

Recommendation

It is recommended to use only mapping instead of array

Note:

The `createERC20Pool` function deploys a new contract every time it's called.

Given that a new contract is deployed with each call, it's essential to note that the state of the new contract, including the allowance, is initialized to zero by default.

Therefore, we don't need to explicitly set the allowance to zero before changing it, as it's already at zero for each newly deployed contract. This means that the vulnerability related to non-zero allowance values doesn't apply in this context.

Status

Acknowledged

[M-09] ArmswapV6ERC20#global variable - The Time Lock delay in the contract is set to a constant value of zero.

Description

The timelock delay is set to zero in the contract. This allows users to bypass the timelock mechanism in the setVault and setMinter functions.

Vulnerability Details

The ArmswapV6ERC20 function sets the delay to 0 days using a constant value, which means that this value cannot be changed in the future.

```
uint public constant DELAY = 0 days;
```

This value is utilised in the `setVault` and `setMinter` functions, allowing users to update the addresses of the minter and vault without any delay.

Impact

This enables the user to keep updating the minter and vault value without any time delay.

Recommendation

Ensure that the DELAY is set to a desired value greater than 0.

Status

Resolved

Low

[L-01] ARMswap#transfer - Dust transfers are permitted

Description

A large number of dust transfers can contribute to network congestion and spam, increasing gas fees.

Vulnerability Details

Spamming the transfer() with dust amount to increase the gas cost.

```
function transfer(address to,uint256 value) public virtual override
requireTokenTransferEnabled(_msgSender()) whenNotBlackListed(_msgSender(),
to) returns (bool) {

    address sender = _msgSender();

    _transfer(sender, to, value);

    return true;

}
```

Impact

Potentially increase gas costs for users.

Recommendation

Enforcing a minimum transaction amount can prevent attackers from clogging the network with zero amount or dust transactions.

Status

Resolved

[L-02] AdminControl.sol#applyAdmin - Event emitting before the function ends

Description

In the function the event “applyAdmin” is emitting before the function ends

Vulnerability Details

The function “applyAdmin” is emitting an “applyAdmin” event before the whole function ends its executing. If due to any reason the code fails to complete and there is any offchain mechanism that is looking for such type of events to perform some type of function then it will wrongfully perform that function cause the event has been emitted already. Also it is against CEI Pattern to emit event before the function ends

```
function applyAdmin() external {

        require(msg.sender == pendingAdmin, "AdminControl: Forbidden");

        emit ApplyAdmin(admin, pendingAdmin);

        admin = pendingAdmin;

        pendingAdmin = address(0);

    }
```

Recommendation:

Make sure that the event is emitted after the whole calculation has been ended.

Status

Resolved

L-03] ArmswapV6ERC20.sol#_mint - Wrong event being emiited

Description

Wrong event is being emitted in several functions

Vulnerability Details

In the "_mint" function we can see that "transfer" event is being emitted which is wrong and should be emitted and "minted" event should be in the place of "transfer" event. Also "_burn" function have same issue

```
function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");


        _totalSupply += amount;
        balanceOf[account] += amount;
        emit Transfer(address(0), account, amount); //@audit wrong event
    }


function _burn(address account, uint256 amount) internal {
             require(account  !=  address(0),  "ERC20:  burn  from  the  zero
address");


        uint256 balance = balanceOf[account];
        require(balance >= amount, "ERC20: burn amount exceeds balance");


        balanceOf[account] = balance - amount;
        _totalSupply -= amount;
        emit Transfer(account, address(0), amount); //@audit wrong event
    }
```

Recommendation:

Make sure to emit the correct event.

Note:

In response to this claim, the ArmSwap team understands the concern regarding the emitted events in the `_mint` and `_burn` functions. However, it's important to note that the `Transfer` event is standard and widely accepted within the ERC-20 token standard. This event has been consistently used from the initial versions of ERC-20 to the newer ones, and it is also utilized by OpenZeppelin, a widely trusted library for smart contract development.

Therefore, we have chosen to retain the `Transfer` event in the `_mint` and `_burn` functions for compatibility and consistency with industry standards.

Status

Acknowledged

Gas:

[G-01] ArmswapV6ERC20.sol - Checking value of constant

Description

Waste of gas due to redundant require condition

Vulnerability Details

There are functions like "_deposit" and "withdraw" that are checking the value of constant "underlyingIsMinted" and make sure that it is not "true" but if we see that "underlyingIsMinted" has been given constant so its value can't be changed and has been initialised with "false" value so these type of require statements are waste of gas

```
function _withdraw(address from, uint amount, address to) internal returns
(uint) {

        require(!underlyingIsMinted);

        require(underlying != address(0) && underlying != address(this));



function _deposit(uint amount, address to) internal returns (uint) {

        require(!underlyingIsMinted);
        require(underlying != address(0) && underlying != address(this));

        _mint(to, amount);

        return amount;

    }
```

Recommendation:

Remove these require condition to save gas.

Status

Resolved

[G-02] ArmswapV6ERC20.sol - First check Balance then proceed to save gas

Description

Gas can be saved by checking user balance first.

Vulnerability Details

In the "_withdraw" function there is a call to "_burn" function and that function is checking if the user has the required balance to withdraw and if he doesn't then the function will revert. Now the issue is that this approach will cost more gas to the user. Instead of this you can check balance in the original "_withdraw" function and if the balance isn't enough then the function will revert.

```
function _burn(address account, uint256 amount) internal {

            require(account != address(0), "ERC20: burn from the zero
address");


        uint256 balance = balanceOf[account];

        require(balance >= amount, "ERC20: burn amount exceeds balance");


        balanceOf[account] = balance - amount;

        _totalSupply -= amount;
```

```
    }
```

Recommendation:

Check user balance first then proceed to burn function

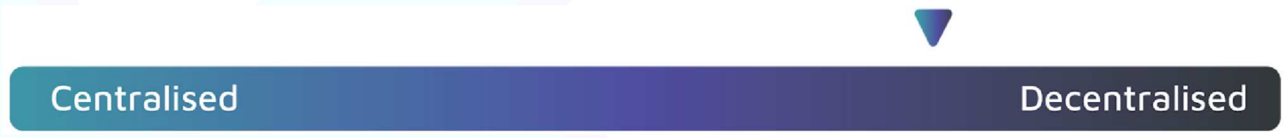
Status

Resolved

Centralisation

The ArmSwap 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 ArmSwap seems to have a sound and well-tested code base, now that our findings have been resolved/acknowledged in order to achieve security. 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 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

Hashlock.