hashlock.

2024

Security Audit

BITL (dApp)

August 2024 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit scope	7
Security Rating	8
Intended Smart Contract Behaviours	9
Code Quality	10
Audit Resources	10
Dependencies	10
Severity Definitions	11
Audit Findings	12
Centralisation	23
Conclusion	24
Our Methodology	25
Disclaimers	27
About Hashlock	28

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE THAT 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 THE USE OF THE CLIENT.

Executive Summary

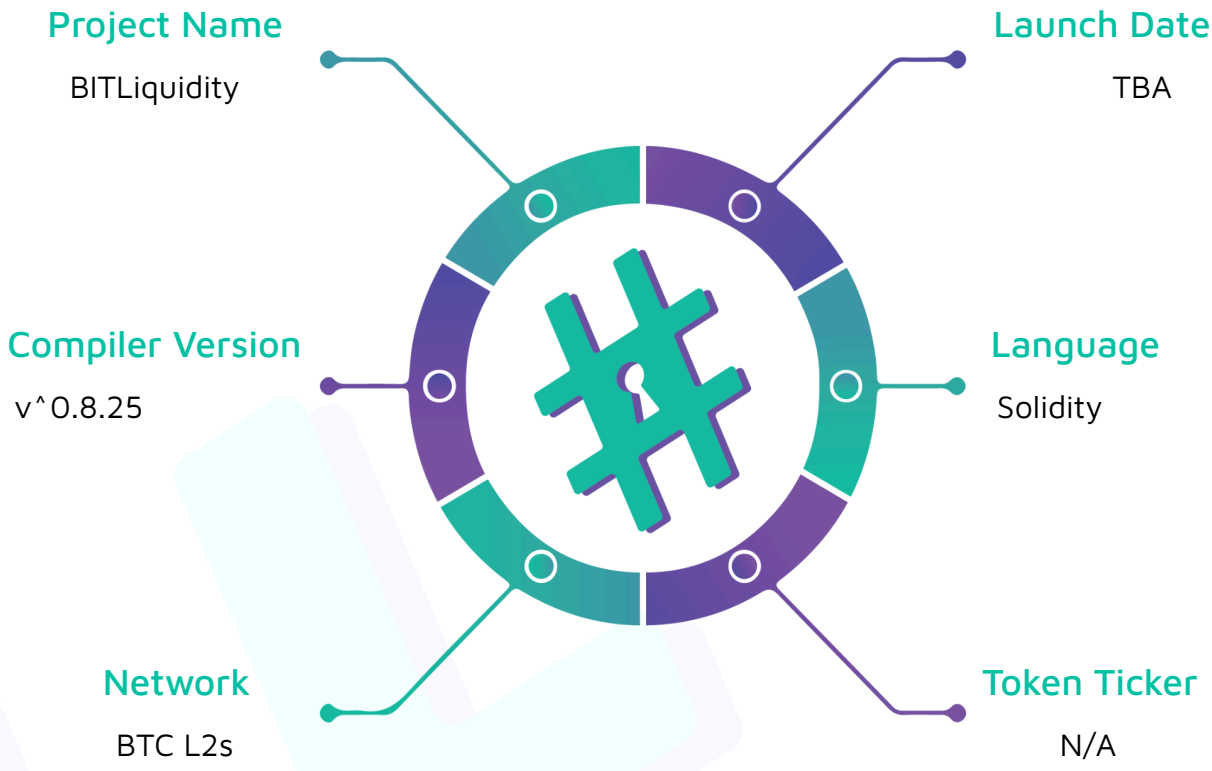
The BITLiquidity team partnered with Hashlock to conduct a security audit of their Dex aggregator smart contracts. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

Project Context

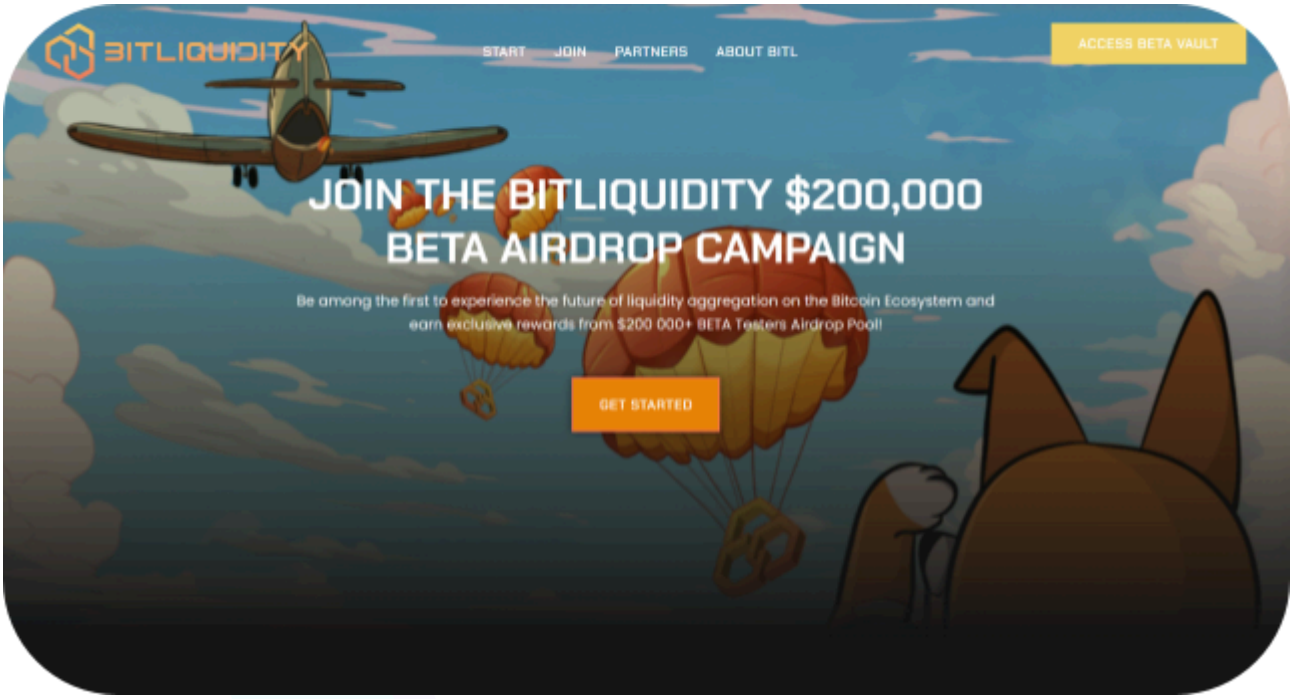
BITLiquidity is a DeFi dApp built on Bitcoin Layer 2 Networks that allows you to perform optimised, multiple swaps via common adapters to exchange tokens and the wrapped native token in both directions.

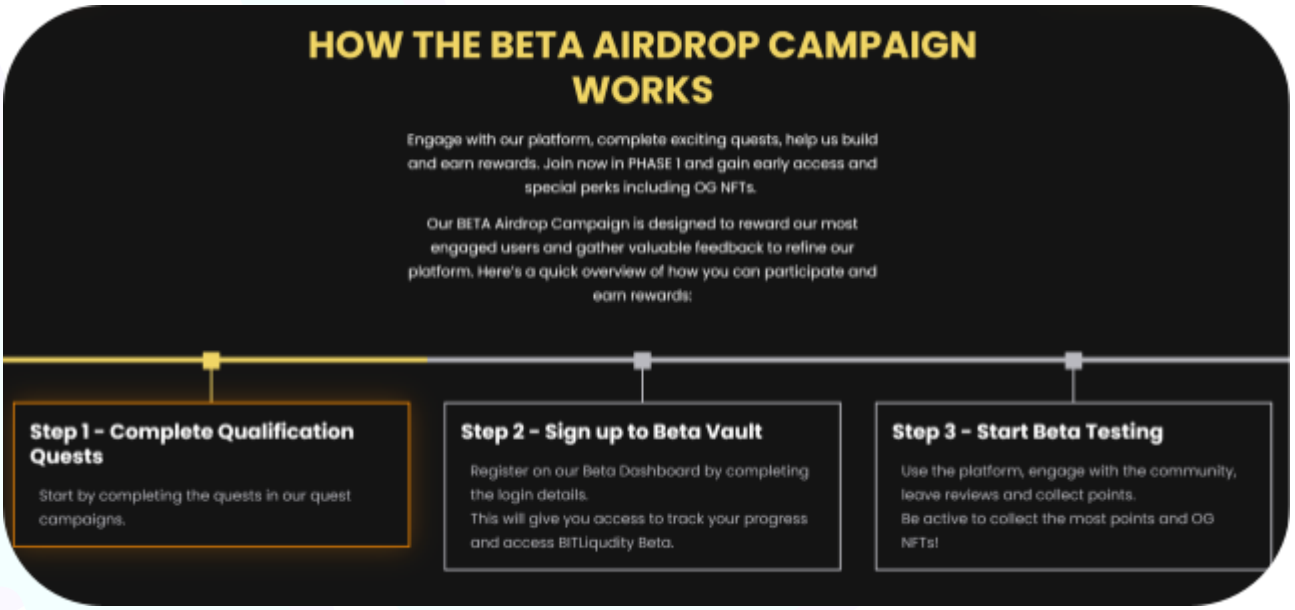
Project Name: BITLiquidity
Compiler Version: ^0.8.25
Website: https://beta.BITLiquidity.io/
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the BITLiquidity 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	**BITLiquidity Smart Contracts**
Platform	**Bitlayer / Solidity**
Audit Date	**August, 2024**
Contract 1	Aggregator.sol
Contract 1 MD5 Hash	786ef77c776ace6c0d691856a0d28af5
Contract 2	BitCowAdapter.sol
Contract 2 MD5 Hash	f251637aabc9353fb8c4c460db2501bd
Contract 3	CurveTwoPoolAdapter.sol
Contract 3 MD5 Hash	68ebebd2ce5200554d4583d958fb3be5
Contract 4	UniswapV2Adapter.sol
Contract 4 MD5 Hash	a769049686156b9df3f8c673cc9396eb
Contract 5	WBTCProxy.sol
Contract 5 MD5 Hash	56c61044f2895729dedef43a53d9e7c7
Contract 6	UniswapV3Adapter.sol
Contract 6 MD5 Hash	4b2a89f7203e4a5f2faf88f855827056

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.

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

Hashlock found:

2 High-severity vulnerabilities

1 Medium-severity vulnerabilities

5 Low-severity vulnerabilities

1 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 Behaviours

Claimed Behaviour	Actual Behaviour
Aggregator.sol - Main entry point for the protocol - Allows to initiate swaps to BTC, from BTC and between other tokens - Relies on Adapters to complete the swaps	**Contract achieves this functionality.**
BitCowAdapter.sol - An adapter contract, callable by Aggregator - Allows swaps via BitCow	**Contract achieves this functionality.**
CurveTwoPoolAdapter.sol - An adapter contract, callable by Aggregator - Allows swaps via Curve	**Contract achieves this functionality.**
UniswapV2Adapter.sol - An adapter contract, callable by Aggregator - Allows swaps via UniswapV2	**Contract achieves this functionality.**
WBTCProxy.sol - Contains only a function to convert of Wrapped Bitcoin (WBTC) to BTC and send it to a given address	**Contract achieves this functionality.**
UniswapV3Adapter.sol - An adapter contract, callable by Aggregator - Allows swaps via UniswapV3	**Contract achieves this functionality.**

Code Quality

This Audit scope involves the smart contracts of the BITLiquidity project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

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 BITLiquidity 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 help understand the overall architecture of the protocol.

Dependencies

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] Usage of transferFrom/transfer is unreliable and may cause issues in the future

Description

The project widely uses `transfer/transferFrom` to send ERC20 tokens to and from the contracts. Using the Ethereum network as a benchmark, it is known that some popular tokens, such as BAT or USDT cause issues with this type of transfers. For example BAT does return false on failure, which is not caught by `transfer` and a failed transfer may still be accounted for, and USDT is not compliant with IERC20 since `transfer` does not return a value, so attempt to transfer always reverts due to interface mismatch.

On Bitlayer, the USDT token is a proper ERC20 standard. However, based on token addresses found in the unit tests, some of them have unverified code therefore it is difficult to assess how they will behave. Considering the router protocol will be dealing with many token types, maximum compatibility should be the target.

Recommendation

Use `safeTransfer / safeTransfer` from the OpenZeppelin libraries to handle token transfers.

Status

Resolved

[H-02] Usage of approve may cause issues with some tokens in the future

Description

Similarly to issue L-01, on the Ethereum network it was known to have issues related to `approve` function. For example, on Ethereum USDT `approve` can be called only if its already zero https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code in order to prevent `approve` frontrunning. Similarly it may be expected that some tokens on Bitlayer might have this requirement in the future.

Since the `approve` is called for the initial amount, and during a swap not 100% of the allowance amount may be consumed, it might happen that such a token will become untradeable for the protocol.

Recommendation

Consider resetting `approve` amount to zero if its other value after a successful swap.

Status

Resolved

Medium

[M-01] UniswapV2Adapter - Insufficient slippage protection due to hard coded block.timestamp as the deadline

Description

The calls to swap operations performed by adapters properly include a minimal amount to receive from a swap. However, to fully protect users from potential slippage, a deadline should be supplied. If the deadline is hard coded as block.timestamp, it effectively means no deadline, since it always will be replaced with the current block, thus it won't change anything. While the minimal amount already protects from slippage to some point, an additional factor of enforcing proper deadline prevents from including the transaction in more favourable conditions for a MEV searcher.

Vulnerability Details

An example occurrence in `UniswapV2Adapter`:

```
    function swap(
        uint inputAmount,
        address receiver,
        Hop[] calldata hops,
        bytes calldata data
    )
        external
        override
        onlyAggregator
        onlyValidInputs(hops, data)
        returns (uint)
    {
        (address[] memory path, uint minAmount) = _getSwapParams(hops);
        IERC20(path[0]).approve(address(router), inputAmount);
        uint[] memory amounts = router.swapExactTokensForTokens(
            inputAmount,
            minAmount,
            path,
            receiver,
            block.timestamp
        );
        return amounts[amounts.length - 1];
    }
```

This is also present in lines 72, 103, 136

Impact

The protocol swaps with that setting may be targeted by MEV searchers due to the possibility of including the swap in any block.

Recommendation

It's recommended that users can specify the deadline themselves.

Status

Acknowledged

Low

[L-01] Multiple contracts - Redundant code - data parameter is never used in the protocol

Description

The protocol uses a data parameter to be passed from Aggregator to the end adapters. However this data is never used anywhere. It is meant to be optional, additional data for the swaps however it turns out it is never used and never passed to the end protocol (Bitcow, Uniswap or Curve).

Example occurrence in `Aggregator.sol`:

```
      function aggregateToBTC(
        uint inputAmount,
        IAdapter.Hop[] calldata hops,
        uint[] calldata indices,
        bytes[] calldata data
    ) external returns (uint amount) {
[...]
```

```
        amount = _aggregate(
            inputAmount,
            AggregatorParams(false, true, false),
            hops,
            indices,
            data
        );
```

The data is further passed to an adapter, for example `UniswapV2Adapter.sol` where data is still unused, there is one function below that takes it as an argument but does not process it:

```
    function _checkInputs(Hop[] memory hops, bytes memory data) internal view {
        uint tmpDex = dex;
        for (uint i; i < hops.length; i++) {
            if (hops[i].dex != tmpDex) {
                revert UnsupportedDex();
            }
            supportedPair(hops[i].pair);
        }
    }
```

Finally, another example occurrence is function `swap` in `UniswapV2Adapter.sol`:

```
    function swap(
        uint inputAmount,
        address receiver,
        Hop[] calldata hops,
```

```
        bytes calldata data
    )
        external
        override
        onlyAggregator
        onlyValidInputs(hops, data)
        returns (uint)
    {
        (address[] memory path, uint minAmount) = _getSwapParams(hops);
        IERC20(path[0]).approve(address(router), inputAmount);
        uint[] memory amounts = router.swapExactTokensForTokens(
            inputAmount,
            minAmount,
            path,
            receiver,
            block.timestamp
        );
        return amounts[amounts.length - 1];
    }
```

The data is not passed anywhere in the end.

Recommendation

Remove unused parts to improve readability and decrease complexity of the code.

Status

Acknowledged

[L-02] Contracts - Usage of transfer for sending native funds may fail if recipient is a smart contract with complex receive logic

Description

The project uses `transfer` to send native funds to the caller. Transfer function forwards hardcoded amount of 2300 gas, and if the admin is using a multisignature wallet or another smart contract receiving funds with a fallback function that contains some logic, the transaction will fail making it unable to transfer the funds out of the contract.

An occurrence in `Aggregator.sol` in function `extractFee`:

```
        if (tokenAddress == address(0)) {
         payable(owner()).transfer(address(this).balance);
         return;
      }
```

Recommendation

Use `call` instead to send native funds.

Status

Resolved

[L-03] Contracts - Usage of Ownable and AccessControl simultaneously introduces unnecessary complexity

Description

The project uses both `Ownable` and `AccessControl` from OpenZeppelin. `Ownable` is a simple library that introduces the role of Owner (single privileged party) while `AccessControl` allows for more granular management of multiple roles of various

privilege levels. The project also introduces custom modifiers to check for role membership.

However this approach seems to be overly complicated, because `AccessControl` alone has sufficient tools to manage this. On the other hand, if there are only two roles in the protocol (Admin and Aggregator) which are not changed frequently, possibly AccessControl is not needed, and a simple ownable role may be sufficient.

The more complexity in the code, the higher chance of overlooking something leading to a security issue. Therefore simplicity should be prioritised. As of now in the Aggregator contract:

- Is Ownable already (the deployer will be owner)
- The Owner is also granted an _ADMIN role in the same time
- Additional modifier onlyAdmin is introduced to check for _ADMIN role
- Some functions use onlyOwner modifier and some onlyAdmin, but this is in fact the same account
- onlyAdmin controls functions such as addAdminRole, but AccessControl supports this natively via function such as grantRole (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol#L122) therefore it is not needed to write your own wrapper for this.

On the other hand, Adapters, for example UniswapV2Adapter has following management:

- Is Ownable
- There are two modifiers onlyAdmin and onlyAggregator but AccessControl already provides onlyRole modifier
- _ADMIN is the same as owner
- _AGGREGATOR has to be added manually as admin and does not play any role aside of address check

Therefore, current access control seems to be overly complex.

Recommendation

Based on current protocol state, it seems sufficient to use just `Ownable` with its onlyOwner modifier available without a need to re-declare it, and a single Owner role for management. A two-step ownership management can be used e.g. OpenZeppelin's Ownable2Step. A simple function such as `setAggregatorAddress(address _aggregator) onlyOwner` and `address public aggregator` variable accompanied by a `onlyAggregator()` modifier is sufficient for Adapters.

Status

Resolved

[L-04] Multiple contracts - Funds may be stuck in contract due to lack of withdraw possibility

Description

There are two types of this issue. First is related to the WBTCProxy contract and the other to Adapters.

WBTCProxy Contract implements a `receive()` handler but does not have the ability to withdraw any funds that are accidentally sent there. On the other hand, sending funds to the contract requires user error, which is an unlikely, but not improbable event. If funds are sent to the contract, they become stuck on it and no one can rescue/access them.

The second occurrence concerns Adapters which are used to forward funds to be swapped on associated DEXes. It may be expected that so-called dusts will remain from token swaps and at some point they may grow to a significant value. However, there is no way to retrieve them.

Additionally, in `BitCowAdapter.sol` line 96

```
    function swapFromBTC(
        address receiver,
        Hop[] calldata hops,
```

```
        bytes calldata data


    )
        external
        payable
        override
        onlyAggregator
        onlyValidInputs(hops, data)
        returns (uint amount)
    {
        (, address lastToken) = _getTokens(hops[hops.length - 1]);
        WBTC.deposit{value: msg.value}();
        uint inputAmount = WBTC.balanceOf(address(this));
```

If WBTC balance is present on the contract which is supposed to be dust anyway (not significant amount, thus still low severity) then an user may try to include it in their swaps.

Recommendation

Remove the receive handler or implement a sweep/rescue() or similar functions allowing the owner to withdraw contract BTC balance.

Consider adding a rescue/sweep type function to collect dusts from Adapters.

In BitCowAdapter.sol, change the inputAmount to be equal to msg.value instead of contract balance.

Status

Resolved

[L-05] Multiple Contracts - Missing events emission

Description

Some key state changing functions do not emit an event associated with the change. Which leads to decreased visibility of key state changes.

The issue occurs in functions:

`WBTCProxy.sol` constructor line 10-12

`Aggregator.sol` constructor 42-45

`setFee` 258-259 in `Aggregator.sol`

Recommendation

Add and emit a suitable event for aforementioned state changing functions.

Status

Resolved

Gas

[G-01] Aggregator - feeDecimals can fit into an u8 instead of u16

Description

`feeDecimals` is used to specify the fee multiplier. Since the maximal value of u8 is 255, it is sufficient to use it instead of u16 to save that value.

Recommendation

Change `u16` to `u8` in this case.

Status

Resolved

Centralisation

The BITLiquidity 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 BITLiquidity 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 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 behavior 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 still need to verify the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown not to 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 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


#Hashlock.