hashlock.

Security Audit

Lilypad (Defi)

March 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

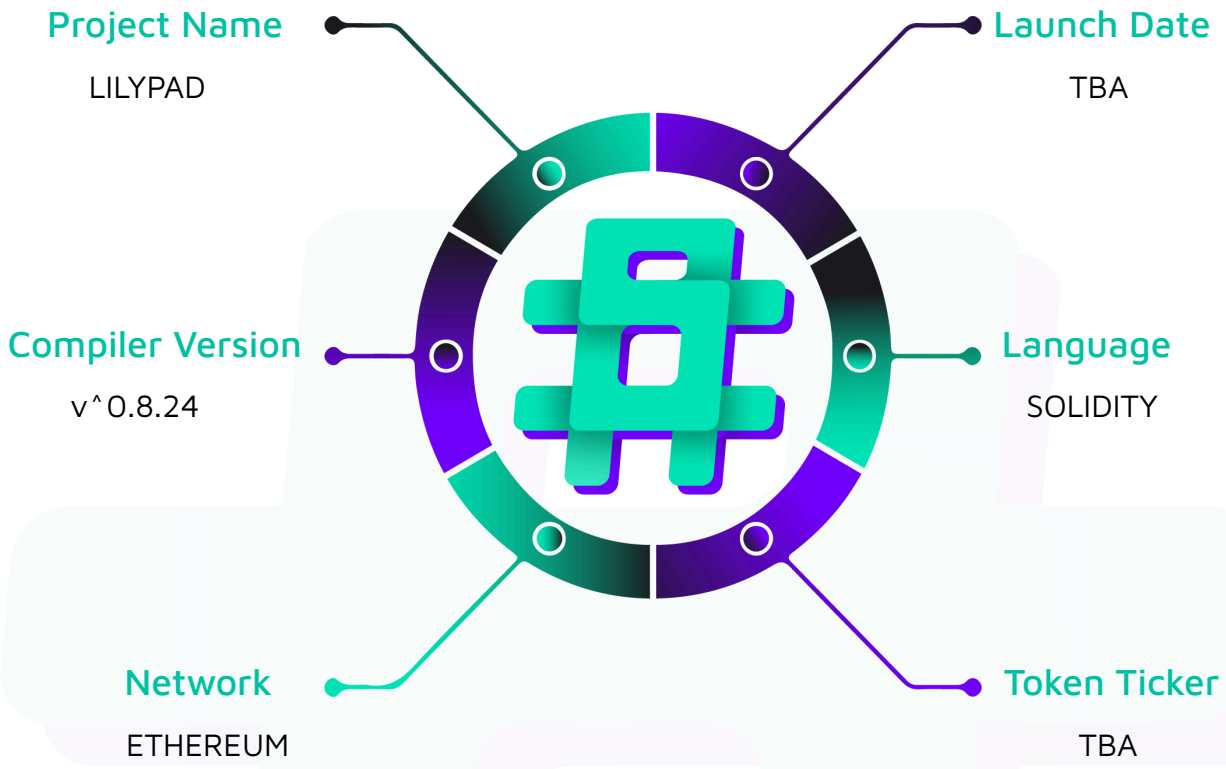
The Lilypad team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

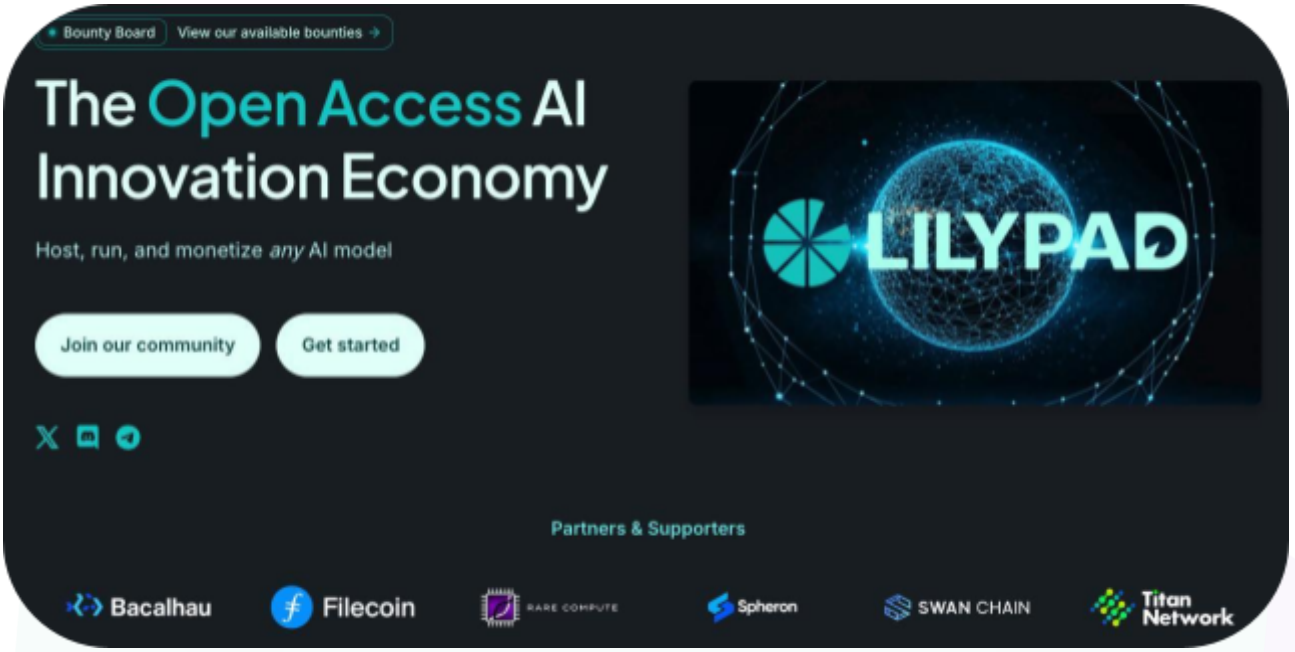
Lilypad Network is building a decentralized, trustless compute network for Web3 applications, leveraging the Bacalhau Project to enable off-chain AI inference, machine learning training, and decentralized science (DeSci). It provides a distributed GPU network for scalable AI workloads and allows smart contracts to access verifiable off-chain computations. Users can contribute computing power as Resource Providers, and developers can create custom job modules. Lilypad fosters community engagement through bounties and an ambassador program. Notable use cases include powering AI-driven applications such as the AI Oncology Agent, demonstrating its capabilities in decentralized healthcare and research. By facilitating decentralized, scalable computing, Lilypad is poised to enhance Web3 infrastructure.

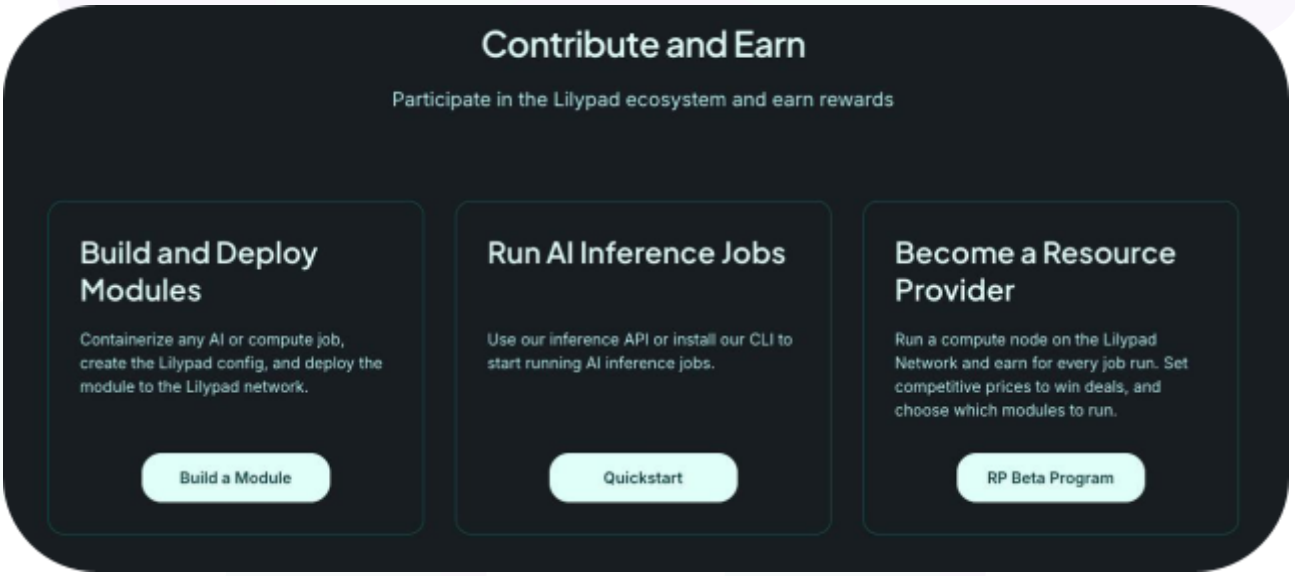
Project Name: Lilypad
Project Type: Defi
Compiler Version: ^0.8.24
Website: www.lilypad.tech
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the Lilypad 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	Lilypad Smart Contracts
Platform	EVM / Solidity
Audit Date	March, 2025
Contract 1	LilypadContractRegistry.sol
Contract 1 MD5 Hash	21f7cd76e030e60a83e87750b68285a9
Contract 2	LilypadModuleDirectory.sol
Contract 2 MD5 Hash	1551ee98f391605cfb7307983f816bf3
Contract 3	LilypadPaymentEngine.sol
Contract 3 MD5 Hash	15243562587395b2b8de92bf00a81ab8
Contract 4	LilypadProxy.sol
Contract 4 MD5 Hash	6bc09bbc9863c8b7ea27577e3d8d7dbc
Contract 5	LilypadStorage.sol
Contract 5 MD5 Hash	adfa2468b2f454442d0330700a0098f5
Contract 6	LilypadToken.sol
Contract 6 MD5 Hash	f08bef2bfd9d832fda57b523d6512c0d
Contract 7	LilypadTokenomics.sol
Contract 7 MD5 Hash	aa775c24825b2a1d6ae36b1350d3351b
Contract 8	LilypadUser.sol
Contract 8 MD5 Hash	42273a8e7e03053fc6ba8347bc670038
Contract 9	LilypadVesting.sol
Contract 9 MD5 Hash	523dc22873374c05deb470090a03f44c
Contract 10	SharedStructs.sol
Contract 10 MD5 Hash	61270545e67d608c3d0c32f1e1924e30
GitHub Commit Hash	465a20ecd1cc62a8b84f055721f11e3b946f70ea

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 have identified some vulnerabilities that need to be addressed prior to launch.

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.

We initially identified some vulnerabilities that have since been addressed.

Hashlock found:

4 High severity vulnerabilities

2 Medium severity vulnerabilities

2 Low severity vulnerabilities

2 Gas Optimisations

8 QA

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
LilypadContractRegistry.sol A contract that stores all the addresses of the contracts in the protocol	**Contract achieves this functionality.**
LilypadModuleDirectory.sol A contract that allows for the creation and management of modules on the network	**Contract achieves this functionality.**
LilypadPaymentEngine.sol A contract that represents the core payment rails of the protocol allowing for escrow deposits and payouts	**Contract achieves this functionality.**
LilypadProxy.sol The main access point for users of the protocol	**Contract achieves this functionality.**
LilypadStorage.sol A contract that allows for the storage of deals, results and validation results on the network	**Contract achieves this functionality.**
LilypadToken.sol A contract that represents the LILY token on the network	**Contract achieves this functionality.**
LilypadTokenomics.sol A contract that stores the tokenomics values used in the protocol	**Contract achieves this functionality.**
LilypadUser.sol A contract that allows for the creation and management of users on the network	**Contract achieves this functionality.**

LilypadVesting.sol A contract that allows for the vesting of tokens on the network of various stakeholders	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Lilypad 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 on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Lilypad 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.

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] LilypadModuleDirectory#transferModuleOwnership - Lack of caller verification in transferModuleOwnership function

Description

The current `transferModuleOwnership` function allows transferring the ownership of a module to a new owner. However, the function does not verify the identity of the caller.

Vulnerability Details

The `transferModuleOwnership` function transfers the ownership of a module to `newOwner` and checks if the `newOwner` is already approved by the previous owner by comparing with `_transferApprovals[moduleOwner][moduleName]`.

If the current owner did not approve any wallets, the value of `_transferApprovals[moduleOwner][moduleName]` is 0 by default.

It means malicious actors could transfer any module's ownership to zero address.

```
function transferModuleOwnership(
    address moduleOwner
    address newOwner,
    string memory moduleName
    string memory moduleUrl
) external override returns (bool) {
    ...
    if (_transferApprovals[moduleOwner][moduleName] != newOwner) {
        revert LilypadModuleDirectory__TransferNotApproved();
    }
    ...
}
```

Proof of Concept

An example of a test that simulates the issue is shown below.

```
function test_TransferModuleToZero() public {
    vm.startPrank(CONTROLLER);
    moduleDirectory.registerModuleForCreator(ALICE, “module1”, “url1”);
    vm.stopPrank();
    vm.startPrank(BOB);
    bool transferSuccess = moduleDirectory.transferModuleOwnership(ALICE, address(0), “module1”, “url1”);
    assertTrue(transferSuccess);
    SharedStructs.Module[] memory aliceModules = moduleDirectory.getOwnedModules(ALICE);
    assertEq(aliceModules.length, 0);
}
```

Impact

Malicious actors could transfer any module’s ownership to the zero address and compromise the system.

Recommendation

Add a caller verification which checks if the caller is the owner of the module.

Status

Resolved

[H-02] LilypadPaymentEngine#payEscrow - Lack of caller verification in payEscrow function

Description

The current `payEscrow` function allows paying an escrow from the payee address with a payment reason. This function allows malicious actors to pay the escrow with unexpected payment reasons from the payee if the contract has any allowance from the payee.

Vulnerability Details

Each payee will need to approve their l2tokens before executing the `payEscrow` function.

Malicious actors could monitor the system and make malicious transactions with the payee address and incorrect payment reasons right after the payee approves.

```
function payEscrow(address _payee, SharedStructs.PaymentReason _paymentReason, uint256 _amount) external moreThanZero(_amount) returns (bool) {
    ...
    bool success = l2token.transferFrom(_payee, address(this), _amount);
    ...
    emit LilypadPayment__escrowPaid(_payee, _paymentReason, _amount);
}
```

Proof of Concept

An example of a test that simulates the issue is shown below.

```
function test_PayEscrowWithIncorrectParameters() public {
    amount = 100 * 10 ** 18;
    vm.startPrank(ALICE);
    token.approve(address(paymentEngine), amount);
    vm.stopPrank();
    vm.startPrank(BOB);
    paymentEngine.payEscrow(ALICE, SharedStructs.PaymentReason.ValidationFee, amount);
    assertTrue(transferSuccess);
}
```

Impact

Malicious actors could execute the function with unexpected payment reasons and break the whole system.

Recommendation

Add a caller verification which checks if the caller has `CONTROLLER_ROLE` .

Status

Resolved

[H-03] LilypadPaymentEngine#withdrawEscrow - Tokens stuck due to role revocation after deposit

Description

The withdrawEscrow only allows resource providers or validators to withdraw their escrows. If the resource providers or validators lose the role after depositing, their escrow would be stuck.

Vulnerability Details

The `withdrawEscrow` function checks if the caller is a resource provider or validator.

```
function withdrawEscrow(address _withdrawer, uint256 _amount)
    external nonReentrant moreThanZero(_amount) returns (bool) {
    ...
    if (lilypadUser.hasRole(_withdrawer, SharedStructs.UserType.ResourceProvider)
      || lilypadUser.hasRole(_withdrawer, SharedStructs.UserType.Validator)
    ) {
        ...
    } else {
        revert LilypadPayment__escrowNotWithdrawableForActor(_withdrawer);
    }
    ...
}
```

These roles could be assigned or revoked by the `CONTROLLER_ROLE` in the `LilypadUser` contract. If a resource provider or validator loses the role after depositing, they can not withdraw and their funds would be stuck in the contract.

Recommendation

Allow any users to withdraw with a time limit if they have escrows locked in the contract.

Status

Resolved

[H-04] LilypadPaymentEngine#handleJobFailure - Division before multiplication may result in precision loss

Description

The handleJobFailure function first divides lilypadTokenomics.resourceProviderActiveEscrowScaler() by 10000 and then multiplies it by the fee amount when calculating resourceProviderRequiredActiveEscrow.

```
uint256 resourceProviderRequiredActiveEscrow = (
                                    deal.paymentStructure.priceOfJobWithoutFees         +
deal.paymentStructure.resourceProviderSolverFee)                                        *
(lilypadTokenomics.resourceProviderActiveEscrowScaler() / 10000);
```

Division before multiplication could result in precision loss.

Recommendation

Perform multiplications before divisions to avoid precision loss.

Status

Resolved

Medium

[M-01] LilypadContractRegistry, LilypadPaymentEngine, LilypadProxy, LilypadTokenomics, LilypadUser - Anyone can initialise the implementation contracts

Description

The above contracts are upgradeable smart contracts.

The implementation contracts allow anyone to call the `initialize()` function due to the absence of a `_disableInitializers()` call in the `constructor`.

Impact

Anyone can take control of the implementation contracts.

Recommendation

It's recommended to add `oz-upgrades-unsafe-allow constructor`.

```
/// @custom:oz-upgrades-unsafe-allow constructor

constructor() {

    _disableInitializers();

}
```

Status

Resolved

[M-02] LilypadPaymentEngine#initialize - l2token decimal is not confirmed to be 18 which all deposits could be reverted depending on the decimals of this token

Description

The `payEscrow` function requires the amount to be deposited to be greater than `MIN_RESOURCE_DEPOSIT_AMOUNT` which is a value with 18 decimal.

It means the `l2token` should have a 18 decimal but the `initialize` function is missing a check to make sure that the `l2token`'s decimal is 18.

Recommendation

Implement a check in the `initialize` function to make sure that the `l2token` decimal is 18.

Note

Lilypad's Team informed Hashlock that this finding is not checked on-chain and does not impact the security or functionality of the contract in practice.
Since the validation of the l2token decimal is managed off-chain, there is no immediate risk of deposits failing due to incorrect decimal assumptions.

Status

Resolved

Low

[L-01] LilypadPaymentEngine, LilypadProxy - Important functions lack event emissions

Description

Critical functions should emit events to provide transparency and enable off-chain monitoring of important state changes. Without events, it becomes difficult to track and verify contract operations, complicating both user interfaces and protocol monitoring. Instances are shown below.

```
function setTreasuryWallet() // LilypadPaymentEngine.sol
function setValueBasedRewardsWallet() // LilypadPaymentEngine.sol
function setValidationPoolWallet() // LilypadPaymentEngine.sol
function setLilypadTokenomics() // LilypadPaymentEngine.sol
function setLilypadUser() // LilypadPaymentEngine.sol
function setLilypadStorage() // LilypadPaymentEngine.sol
function setL2Token() // LilypadPaymentEngine.sol
function setStorageContract() // LilypadProxy.sol
function setPaymentEngineContract() // LilypadProxy.sol
function setUserContract() // LilypadProxy.sol
function setL2LilypadTokenContract() // LilypadProxy.sol
```

Recommendation

Review your contracts to implement event emissions for critical state variable updates.

Status

Resolved

[L-02] LilypadUser#removeRole - Inefficient array length access in loop

Description

The removeRole function reads the length of the validatorAddresses array from storage during every iteration of a for loop. This practice is inefficient because accessing storage is significantly more expensive in terms of gas costs compared to accessing memory. Each time the array length is read from storage, it incurs additional gas costs, which can accumulate and make the function unnecessarily expensive to execute, especially for large arrays.

Recommendation

To optimize gas usage, store the array length in a local variable before the loop begins. This ensures that the length is read from storage only once, and subsequent iterations use the cached value from memory, which is much cheaper to access.

Status

Resolved

Gas

[G-01] LilypadVesting#releaseTokens - State variable that is used multiple times in a function should be cached

Description

The schedule.beneficiary value is read multiple times from storage in the function.
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.

Recommendation

Change the function to cache the state variable that is used multiple times.

Status

Resolved

[G-02] LilypadVesting - State variable that could be made immutable

Description

The l2LilypadToken variable is only set in the constructor and never changed anymore. Such variables could be made immutable to save gas when reading them.

Recommendation

Make the `l2LilypadToken` variable immutable.

Status

Resolved

QA

[Q-01] Contracts - Use of floating pragma

Description

Contracts use a pragma statement that does not specify a fixed compiler version but instead allows the contract to be compiled with any compatible 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

Acknowledged

[Q-02] LilypadContractRegistry - Unused variable declaration

Description

The `version` variable is declared but it's not used in the contract.

Recommendation

Remove the unused variable.

Status

Resolved

[Q-03] LilypadContractRegistry - Unused error declaration

Description

The `LilypadContractRegistry__NotController` error is declared but it's not used in the contract.

Recommendation

Remove the unused error declaration.

Status

Resolved

[Q-04] LilypadToken#constructor - Redundant assignment of a default value

Description

The `alpha` variable is explicitly set to 0 in the constructor.

However, since 0 is the default value for this variable, this assignment is redundant.

Recommendation

Remove the redundant assignment of a default value.

Status

Resolved

[Q-05] LilypadPaymentEngine#payEscrow - Typo in the description of depositTimestamps[_payee] calculation

Description

The description of `depositTimestamps[_payee]` calculation has the following typo.

`withdrawl` -> `withdrawal`

Recommendation

Fix the typo.

Status

Resolved

[Q-06] LilypadModuleDirectory#transferModuleOwnership - currentOwnerModules[moduleIndex] update could be inside if condition

Description

`currentOwnerModules[moduleIndex]` could only be updated when `moduleIndex` is not equal to `currentOwnerModules.length`.

```
function transferModuleOwnership(
    address moduleOwner
    address newOwner,
    string memory moduleName
    string memory moduleUrl
) external override returns (bool) {
    ...
    if (moduleIndex != currentOwnerModules.length - 1) {
        ...
    }
     currentOwnerModules[moduleIndex] = currentOwnerModules[currentOwnerModules.length -
1];
    ...
}
```

Recommendation

Move the `currentOwnerModules[moduleIndex]` update inside if condition.

Status

Acknowledged

[Q-07] LilypadTokenomics#setVValues - Duplicate if conditions

Description

The `setVValues` function has duplicate if checks for the same purpose.

```
function setVValues(uint256 _v1, uint256 _v2) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (_v1 <= _v2) revert LilypadTokenomics__V1MustBeGreaterThanV2();
    if (_v2 >= _v1) revert LilypadTokenomics__V2MustBeLessThanV1();
    ...
}
```

Recommendation

Remove one of the duplicate checks.

Status

Resolved

[Q-08] SharedStructs - Unused struct

Description

The `PaymentDirection` struct is declared but has never been used.

Recommendation

Remove the unused struct.

Status

Resolved

Centralisation

The Lilypad 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 Lilypad 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 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