hashlock.

Security Audit

Immersve (CeDeFi)

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

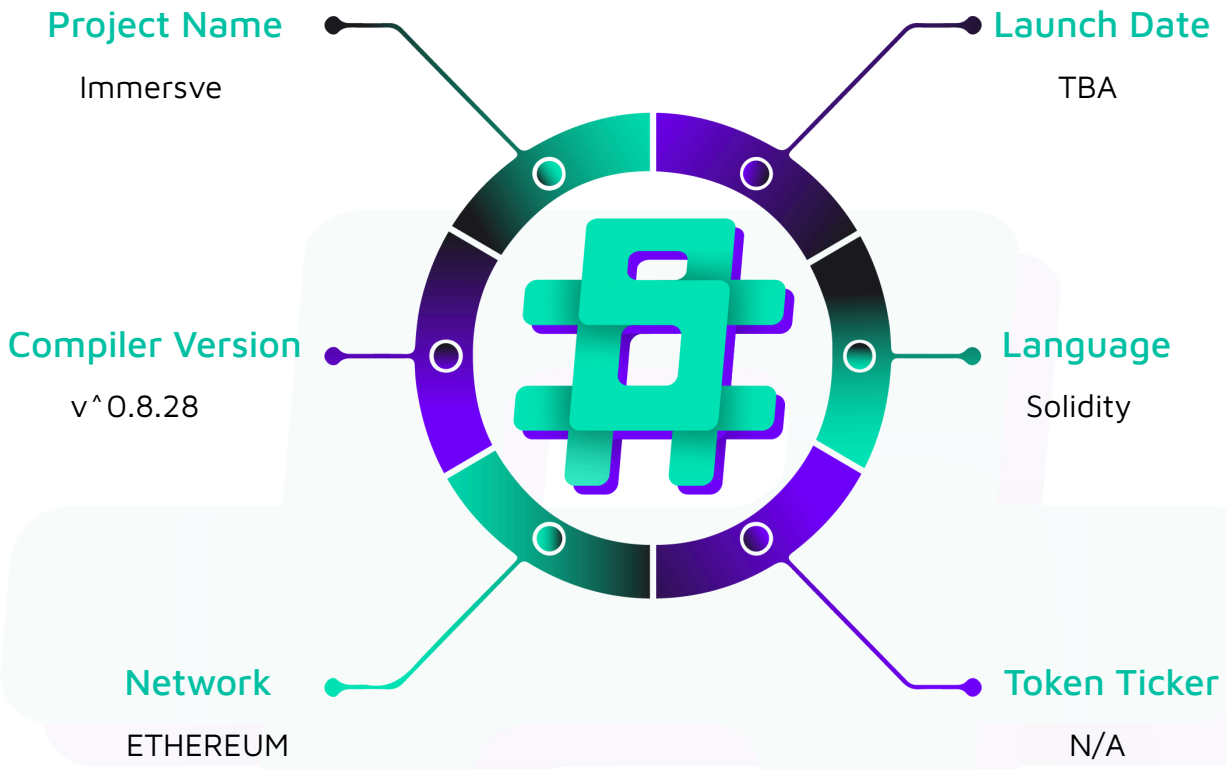
The Immersve team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

Immersve, as principal member of the Mastercard network, uniquely supports both centralised and decentralised payment experiences. Their technology enables users to spend digital cash, including cryptocurrencies, at any Mastercard-accepting merchant, whether online, in physical stores, or in the metaverse. Through the use of their smart contracts, Immersve ensures user control over funds while facilitating seamless transactions across various platforms.

Project Name: Immersve
Project Type: CeDefi
Compiler Version: ^0.8.28
Website: https://immersve.com
Logo:

Visualised Context:



Project Visuals:

Audit Scope

We at Hashlock audited the solidity code within the Immersve 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	**Immersve Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**March, 2025**
Contract 1	FundsManagerLogic.sol
Contract 1 MD5 Hash	053cadb958c896c914ba4d73ab2eecb0
Contract 2	FundsStorageLogic.sol
Contract 2 MD5 Hash	6294401aaa19b0e69d699ba133fb828d
GitHub Commit Hash	5acb8f807c600dede827c7b5baad79a5385e401c
Fix Review GitHub Commit Hash	25ee3a66bb79bbc4e9b2d1776935ddd2f82e5d1c

Note: *Hashlock initially audited the code associated with the "Audited GitHub Commit Hash" and the findings presented in this report pertain specifically to that commit. For the "Fix Review GitHub Commit Hash", Hashlock solely verified the status of the identified findings and did not conduct a comprehensive review to assess any additional code implementations.*

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 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. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

8 Medium severity vulnerabilities

4 Low severity vulnerabilities

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
FundsManagerLogic.sol - Deploys FundsStorage beacon proxy contracts - Handles settlements to Circle and user refunds	**Contract achieves this functionality.**
FundsStorageLogic.sol - Handle user deposits and withdrawals. - Holds tokens	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Immersve 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 Immersve project smart contract code in the form of Github access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

Medium

[M-01] FundsManagerLogic#createFundsStorage - Missing whenNotPaused Modifier in createFundsStorage Function

Description

The `createFundsStorage` function in the `FundsManagerLogic` contract lacks the `whenNotPaused` modifier, which is inconsistent with other state-modifying functions in the contract and could lead to security issues during emergency situations.

Vulnerability Details

The `whenNotPaused` modifier should be applied to functions that modify contract state, interact with external contracts, or perform critical business operations that should be halted during emergency situations. This modifier ensures that these operations cannot be executed when the contract is in a paused state, which is a critical security feature.

```
function createFundsStorage ( address token, string calldata name, FundingMode fundingMode ) external onlyProxy returns (address) {

// ... existing code ...

……
```

In the `FundsManagerLogic` contract, most state-modifying functions like `settle`, `addStorageLiquidity`, `directSpendDebit`, and others correctly implement the `whenNotPaused` modifier. However, the `createFundsStorage` function, which creates new funds storage instances and modifies contract state, does not include this modifier.

The `createFundsStorage` function:

`Creates` new contract instances via BeaconProxy

1. Initializes these instances with critical parameters

2. Updates the contract's state by registering new instances in the `_fundsStorageInstances` mapping
3. Emits events that external systems may rely on

If the contract is paused due to a security incident, vulnerability, or other emergency, users would still be able to create new funds storage instances, potentially exposing these new instances to the same security issues that triggered the pause.

Impact

The absence of the `whenNotPaused` modifier in the `createFundsStorage` function undermines the contract's emergency pause mechanism. During critical security incidents when the contract is paused, users can still create new funds storage instances, potentially exposing them to the same vulnerabilities that triggered the emergency pause.

Recommendation

It is recommended to add the `whenNotPaused` modifier to the `createFundsStorage` function to ensure it cannot be called when the contract is paused:

Status

Resolved

[M-02] `FundsManagerLogic#initialize` - Configurations Reset During Upgrades

Description

The `FundsManagerLogic` contract's `initialize()` function, which is called after every upgrade, unconditionally resets two critical configuration parameters: `_directSpendEnabled` and `_directSpendReversalCutoffSeconds`. This causes direct spend functionality to be disabled after every upgrade and reverts any custom reversal cutoff period to 7 days.

Vulnerability Details

The `initialize()` function in `FundsManagerLogic` is called after every contract upgrade, but it unconditionally resets critical configuration parameters regardless of their previous values:

```
function initialize(
    string calldata commitId,
    string calldata buildNumber
) public onlyProxy onlyRole(DEFAULT_ADMIN_ROLE) {
    /*
     * Warning: this initializer must guard against reinitialization!
     * ...
     * It is the duty of this initializer to do nothing when the proxy state is
     * already initialized.
     */
    bool _proxiesInitialized = address(_fundsStorageBeacon) != address(0);
    if (!_proxiesInitialized) {
        _initialize();
    }
    _buildNumber = buildNumber;
    _commitId = commitId;
    _directSpendEnabled = false; // defaults to disabled - ALWAYS RESET
    _directSpendReversalCutoffSeconds = 7 days; // ALWAYS RESET
}
```

The code checks if proxies are initialized and only calls `_initialize()` if needed, but it unconditionally resets two critical configuration parameters:

1. `_directSpendEnabled` - Always set to false after every upgrade

2. `_directSpendReversalCutoffSeconds` - Always reset to 7 days

    - `_directSpendEnabled` - Always set to false after every upgrade
    - `_directSpendReversalCutoffSeconds` - Always reset to 7 days

This violates the function's own comment that states `"It is the duty of this initializer to do nothing when the proxy state is already initialized."`

Proof Of Concept

1. Administrators enable direct spend functionality and set a custom cutoff period:

2. Contract is upgraded

3. After the upgrade:

- Direct spend is automatically disabled (`_directSpendEnabled` = false)

- Cutoff period is reset to 7 days (`_directSpendReversalCutoffSeconds` = 7 days)

- Users experience transaction failures with "DirectSpendDisabled" errors

- Any in-progress operations suddenly use the changed cutoff period

- Administrators must manually re-enable and reconfigure these settings

Impact

The unexpected reset of configuration parameters during upgrades causes immediate user-facing disruption as direct spend functionality is disabled and security settings revert to defaults without warning. This may result in transaction failures, potentially affecting financial operations.

Recommendation

Modify the `initialize()` function to only set default values during first-time initialization, as there is a setter function if needed

Status

Resolved

[M-03] FundingManagerLogic#_createFundsStorage - Attacker Can Create Duplicate Contract Names Hindering Debugging and Enabling Legitimate Contract Impersonation

Description

The documentation for the `createFundsStorage` function explicitly states that contract names must be "globally unique," but the implementation does not enforce this requirement. This creates a discrepancy between documented behavior and actual implementation.

Vulnerability Details

The `FundsManagerLogic` contract's `createFundsStorage` function accepts a `name` parameter that is documented as requiring global uniqueness across all contract instances. According to the documentation:

> "The name of the deployed contract. This is used only for debugging purposes and must be globally unique. Using your app name, token name, and 'liveness' in the name are recommended, for example: 'Acme Wallet USDC Live'."

However, the implementation only enforces uniqueness on a per-user basis through the CREATE2 salt calculation, which is derived from both the name and the caller's address:

```
uint256 bitmask = 0xffffffffffffffffffffffff0000000000000000000000000000000000000000;

uint256 nameHash = uint256(keccak256(bytes(name)));

uint256 salt = (nameHash & bitmask) + uint160(msg.sender);
```

Due to this implementation, attackers or malicious actors can create storage contracts with identical names without any restriction or error. The contract will deploy successfully in both cases, with the only difference being the deployed contract addresses due to different `msg.sender` values in the salt.

Impact

The primary stated purpose of the name parameter is for debugging, but without global uniqueness, debugging becomes significantly more difficult as multiple contracts share identical identifiers. Also, malicious actors could create contracts with names identical to well-known or trusted entities (e.g., "Coinbase USDC Wallet Live"), potentially misleading users or external systems.

Recommendation

Create a mapping to track used names and require that each name is unique before creating a new funds storage contract. Add a simple check in the createFundsStorage function to enforce the documented global uniqueness requirement:

```
mapping(string => bool) private _usedNames;

// In createFundsStorage function:

require(!_usedNames[name], "Name already in use");

// ... existing code ...

_usedNames[name] = true;
```

Status

Resolved

[M-04] FundsStorageLogic#initialize - Missing ReentrancyGuardUpgradeable Initialization in FundsStorageLogic

Description

The `FundsStorageLogic` contract inherits from `ReentrancyGuardUpgradeable` but fails to call its initialization function in the contract's initialize method. This omission violates upgradeable contract best practices and could potentially compromise the reentrancy protection mechanism, especially during the future upgrade

Vulnerability Details

The `FundsStorageLogic` contract inherits from multiple upgradeable contracts, including `ReentrancyGuardUpgradeable`:

```
contract FundsStorageLogic is IFundsStorage, ReentrancyGuardUpgradeable, EIP712Upgradeable {

    // ... code ...

}
```

However, in its initialize function, it only initializes the EIP712 component while neglecting to initialize the `ReentrancyGuard`:

```
function initialize(address adminAddress, address token, string calldata name, FundingMode fundingMode) public initializer {

 _fundsAdmin = IFundsAdmin(adminAddress);  _token = IERC20(token);

  _name = name;

  _fundingMode = fundingMode;

  /*

   * Initialize the EIP-712 domain separator used when verifying withdraw signatures.

   */

  __EIP712_init("Immersve.FundsStorageLogic", "1");

  // Missing: __ReentrancyGuard_init();

}
```

The `ReentrancyGuardUpgradeable` contract's initialization is crucial as it sets the internal `_status` variable to `NOT_ENTERED`, which is essential for the proper functioning of the `nonReentrant` modifier

Impact

The missing initialization creates several security risks: the contract currently relies on default storage values being zero (which matches `NOT_ENTERED`), but this

implementation detail could change in future OpenZeppelin versions; the `nonReentrant` modifier used in the withdraw function might not provide the expected reentrancy protection; the contract may become incompatible with future upgrades that have different initialization requirements; and this violation of the established upgradeable contract pattern could lead to unforeseen issues when maintaining or upgrading the contract.

Recommendation

Explicitly initialize the `ReentrancyGuard` by adding the missing initialization call

Status

Resolved

[M-05] FundsManagerLogic._adminLogicAddress - Misleading Variable Naming and Documentation for adminLogicAddress

Description

The `FundsManagerLogic` contract sets `_adminLogicAddress` to its own address (address(this)) in the constructor, while the variable name and comments suggest it should reference a separate `FundsAdminLogic` implementation contract. This creates a disconnect between the variable's name/documentation and its actual usage, which could lead to confusion during code review, maintenance, or future upgrades.

Vulnerability Details

In the `FundsManagerLogic` contract, the `_adminLogicAddress` variable is assigned to address(this) in the constructor, indicating the contract handles its own admin functionality:

```
/**
 * The FundsAdminLogic implementation contract.
 * This storage slot will NOT be initialized on a proxy.
 */
address internal _adminLogicAddress;
```

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    // invoked in the context of the contract being deployed
    _disableInitializers();
    _storageLogicAddress = address(new FundsStorageLogic());
    _adminLogicAddress = address(this); // Sets admin logic to self
}
```

This creates several inconsistencies:

1. The variable name `_adminLogicAddress` suggests it should reference a separate `FundsAdminLogic` contract, not itself (as it imports `IFundsAdmin` Interface)
2. The documentation comment reinforces this expectation by referring to "The `FundsAdminLogic` implementation contract"
3. The contract name "`FundsManagerLogic`" implies it only handles manager functionality, not admin functionality
4. The contract actually implements both `IFundsStorageFactory` (manager) and `IFundsAdmin` (admin) interfaces

Impact

The misleading variable naming and documentation in `FundsManagerLogic` creates confusion about the contract's architecture, increasing the risk of errors during maintenance and future upgrades. Developers may incorrectly assume a separate admin implementation exists, potentially leading to implementation mistakes or security vulnerabilities.

Recommendation

Since `FundsManagerLogic` implements both management and admin functionality in a single contract, I recommend:

```
/**
```

```
    * Reference to this contract for admin functionality.
     * FundsManagerLogic implements IFundsAdmin directly rather than using a separate contract.
    * This storage slot will NOT be initialized on a proxy.
    */
   address internal _managerAdminAddress;
```

Status

Resolved

[M-06] FundsManagerLogic#setSettlementAddress - Admin Can Permanently Disable Token Settlements by Setting Zero Settlement Address

Description

The `FundsManagerLogic` contract allows the admin to set token settlement addresses to zero, permanently breaking the settlement functionality for all storage proxies using that token. Once a settlement address is zeroed, users' funds can become permanently locked, as all attempts to settle will revert with `TokenNotSupported`.

Vulnerability Details

The vulnerability stems from two interconnected design issues:

1. Storage proxies are permanently bound to a specific token upon creation with no update mechanism:

```
function initialize(address adminAddress, address token, string calldata name, FundingMode fundingMode) public initializer {
  _fundsAdmin = IFundsAdmin(adminAddress);
  _token = IERC20(token);
  // ... more code ...
}
```

2. The admin can set a token's settlement address to zero with no validation:

```
function setSettlementAddress(address token, address settlementAddress) external onlyRole(DEFAULT_ADMIN_ROLE) {

_tokenSettlementAddress[token] = settlementAddress;

}
```

When a storage proxy attempts to transfer funds to the settlement address, it requires a non-zero settlement address:

```
function transferToSettlementAddress(uint256 amount) external {

  _requireFundsAdmin(msg.sender);

  address settlementAddress = _fundsAdmin.getSettlementAddress(address(_token));

   if(settlementAddress == address(0)) revert TokenNotSupported({ token: address(_token) });

  SafeERC20.safeTransfer(_token, settlementAddress, amount);

}
```

If the settlement address is set to zero, all `settle()` calls will revert, permanently breaking the settlement functionality with no recovery path.

Consider the below example :

- Admin sets USDC settlement address to a valid address (`0xSettlementAddress`)
- Users create multiple storage proxies for USDC through `createFundsStorage`(USDC, "User USDC Storage", FundingMode.STANDARD)
- Users deposit USDC to their storage proxies
- Admin calls `setSettlementAddress`(USDC, address(0)) (accidentally or maliciously)
- All `settle()` calls for USDC storage proxies now revert with `TokenNotSupported`
- Users' funds are locked in storage proxies

Impact

This vulnerability creates a single point of failure where an admin can inadvertently or maliciously lock all user funds across storage proxies for a specific token. Once

triggered, funds are permanently inaccessible as no recovery mechanism exists. The issue can affect multiple users simultaneously with a single transaction, requires only standard admin permissions, and bypasses all existing withdrawal mechanisms.

Recommendation

Add validation to prevent setting settlement addresses to zero:

```
function setSettlementAddress(address token, address settlementAddress) external onlyRole(DEFAULT_ADMIN_ROLE) {
  require(settlementAddress != address(0), "Settlement address cannot be zero");
  _tokenSettlementAddress[token] = settlementAddress;
}
```

Status

Resolved

[M-07] FundsStorageLogic.withdraw - Critical Functions in FundsStorageLogic Lack Emergency Pause Controls

Description

While `FundsManagerLogic` implements pause functionality for emergency scenarios, its companion `FundsStorageLogic` contract lacks this critical security feature. This inconsistency allows users to continue interacting with storage contracts even when the system is meant to be paused during emergencies.

Vulnerability Details

The `FundsStorageLogic` contract contains critical functions like `withdraw()` that lack the `whenNotPaused` modifier:

```
function withdraw(uint256 amount,uint256 expiryDate,uint256 nonce,bytes memory _signature) external nonReentrant { // No whenNotPaused modifier
    _requireFundingMode(FundingMode.DEPOSIT);
    // ... remaining implementation ...
```

```
}
```

This architectural inconsistency means that even when the system is paused, users can still interact directly with storage contracts to execute withdrawals and other operations, which may undermine the intended security controls during emergency scenarios.

Impact

The lack of pause functionality in `FundsStorageLogic` undermines the system's emergency controls, allowing users to continue withdrawing funds and performing critical operations even when the protocol is meant to be frozen during security incidents. This architectural inconsistency creates a bypass of intended safety mechanisms and could prevent effective response to security events.

Recommendation

Inherit from OpenZeppelin's PausableUpgradeable in `FundsStorageLogic` and add the `whenNotPaused` modifier to critical functions like `withdraw()`.

Status

Resolved

[M-08] FundsManagerLogic - Missing onlyProxy Modifier in Multiple Functions Leads to Confusing State Updates And Reverts

Description

Multiple critical functions in the `FundsManagerLogic` contract lack the `onlyProxy` modifier. While some functions will fail when called directly due to dependent contract checks, others will silently update the implementation contract's state instead of the proxy's state, leading to confusion and potential issues.

Vulnerability Details

Here are some of the functions lacking the `onlyProxy` modifier:

```
// Admin functions
```

```
function setSettlementAddress(
    address token,
    address settlementAddress
) external onlyRole(DEFAULT_ADMIN_ROLE) { // Missing onlyProxy
    _tokenSettlementAddress[token] = settlementAddress;
}

// Role management functions
function transferDefaultAdminRole(
    address adminAccount
) external onlyRole(DEFAULT_ADMIN_ROLE) { // Missing onlyProxy
    _grantRole(DEFAULT_ADMIN_ROLE, adminAccount);
    _revokeRole(DEFAULT_ADMIN_ROLE, msg.sender);
}

// Settler role functions
function settle(
    address from,
    uint256 amount,
    uint256 nonce,
    bytes32 merkleRoot
) external onlyRole(SETTLER_ROLE) whenNotPaused { // Missing onlyProxy
    // ... implementation
}
```

This creates two types of issues:

1. Functions like `setSettlementAddress` will silently update the implementation contract's state instead of the proxy's state
2. Functions like settle will appear callable but fail when trying to interact with `FundsStorage` due to proxy checks

Impact

1. Potential security risks if admin assumes state changes took effect in proxy but actually occurred in implementation
2. Confusion during contract interactions when functions appear callable but fail

Recommendation

Add the missing `onlyProxy` modifier, including state modifying functions, to ensure consistent behavior

Status

Resolved

Low

[L-01] Contracts - Use of floating pragma

Description

The contracts `FundsManagerLogic` and `FundsStorageLogic` use a floating pragma version (^0.8.28). It's crucial to compile and deploy contracts with the same compiler version and settings used during development and testing. Fixing the pragma version prevents compilation with different versions. Using an outdated pragma version could introduce bugs that negatively affect the contract system, while newly released versions may have undiscovered security vulnerabilities.

Recommendation

Change the pragma statement in both contracts to a fixed version, such as pragma solidity 0.8.28;. This locks the contract to a specific compiler version.

Status

Resolved

[L-02] Contracts - Missing Initializer Calls for Inherited Upgradeable Contracts

Description

The `FundsManagerLogic` contract inherits from multiple OpenZeppelin upgradeable contracts but does not call all the recommended initializer functions during its initialization process. The contract calls `PausableUpgradeable.__Pausable_init()` in its `_initialize()` function but omits the initializers for `AccessControlUpgradeable` and `UUPSUpgradeable`.

https://forum.openzeppelin.com/t/is-it-mandatory-to-call-pausable-init-or-any-init-functions-within-upgradable-contracts-what-happens-if-you-dont/40565

According to OpenZeppelin's official guidance: "From your initializer, you should always call the `__{ContractName}_init` functions of the parent contracts that you inherit. These are important to call because they initialize those contracts and their linearized parent

contracts, if any. If you don't, some contracts may not be initialized." Even when it doesn't make a functional difference in specific cases, "as a best practice, we recommend calling `__{ContractName}_init` functions for all directly inherited contracts." [1] Not following this established pattern represents a deviation from recommended practices for upgradeable contracts, which could lead to unexpected behavior if future versions of the libraries change or if the inheritance structure is modified.

Recommendation

For code consistency and following OpenZeppelin's best practices, call all required initializer functions for inherited contracts

Status

Resolved

[L-03] Contracts - Unbounded DirectSpend Reversal Window Can Be Set to Unsafe Values

Description

The `_directSpendReversalCutoffSeconds` variable determines how long users have to reverse a direct spend transaction after it's been executed. This critical parameter defines the dispute window during which transactions can be undone, defaulting to 7 days. The issue is that the setter function lacks any bounds checking:

```
function setDirectSpendReversalCutoffSeconds(

    uint256 expiry

) external onlyRole(DEFAULT_ADMIN_ROLE) {

    _directSpendReversalCutoffSeconds = expiry;

}
```

This unbounded configuration creates several risks:

1. If set to zero or a very low value, users would have no meaningful opportunity to reverse erroneous transactions, effectively disabling a critical consumer protection feature.
2. If set to an extremely high value (e.g., years), it would create indefinite uncertainty about transaction finality, causing accounting issues and potentially violating reasonable expectations about when transactions become final.

While the default value of 7 days is reasonable, nothing prevents an admin from changing this to an unsafe value.

Recommendation

Implement reasonable minimum and maximum bounds on the reversal window.

Status

Resolved

[L-04] FundsManagerLogic#settle - No Amount Validation in settle Function Allows Full Contract Drain

Description

The settle function in the `FundsManagerLogic` contract allows authorized settlers to transfer arbitrary amounts of tokens from a funds storage contract to a settlement address without any validation on the amount. This lack of validation means a settler can transfer any amount - including the entire balance of a storage contract - either maliciously or accidentally.

Vulnerability Details

The settle function in `FundsManagerLogic.sol` allows a user with the `SETTLER_ROLE` to transfer tokens from a storage contract to a settlement address:.

```
function settle(address from, uint256 amount, uint256 nonce, bytes32 merkleRoot) external
onlyRole(SETTLER_ROLE) whenNotPaused {

    // ... validation code ...
```

```
    fundsStorage.transferToSettlementAddress(amount);

    // ... state updates ...

}
```

This function calls into the `transferToSettlementAddress` function in the `FundsStorageLogic` contract :

```
function transferToSettlementAddress(uint256 amount) external {

   _requireFundsAdmin(msg.sender);

  address settlementAddress = _fundsAdmin.getSettlementAddress(address(_token));

 if(settlementAddress == address(0)) revert TokenNotSupported({ token: address(_token)
});

    SafeERC20.safeTransfer(_token, settlementAddress, amount);

}
```

Here, the issue is that the function does not include: the process involves validating that the amount is both reasonable and expected, verifying that it matches the anticipated settlement value, enforcing upper limits on how much can be transferred in a single operation, and conducting balance checks to ensure the amount is appropriate for the specific funds context.

Impact

A malicious actor with the `SETTLER_ROLE` could drain the entire balance of any funds storage contract in a single transaction. Even honest settlers could make catastrophic mistakes (e.g., entering extra zeros) with no safeguards to prevent such errors.

Recommendation

Add a mechanism where expected settlement amounts must be pre-recorded before they can be executed.

Status

Acknowledged

Centralisation

The Immersve 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 Immersve 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