ENTERSOFT

DeFa

Smart Contract Audit

DATE: 23rd October 2025 | PREPARED BY: Entersoft Singapore

Contents

Revision History & Version Control

Start Date	End Date	Author	Comments/Details
22 September 2025	22 October 2025	Gurkirat, Shashank Wangkar and Laxmi Prasad	Interim Release for the Client

Reviewed by	Released by
Nishita Palaksha	Nishita Palaksha

Entersoft was commissioned to perform a source code review on Defa smart contracts. The review was conducted between September 22, 2025, to October 23, 2025. The report is organized into the following sections.

- **Executive Summary:** A high-level overview of the security audit findings.
- **Technical analysis:** Our detailed analysis of the Smart Contract code

The information in this report should be used to understand overall code quality, security, correctness, and meaning that code will work as described in the smart contract.

1.0 Disclaimer

This is a limited audit report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to: (i) smart contract best coding practices and vulnerabilities in the framework and algorithms based on white paper, code, the details of which are set out in this report, (Smart Contract audit). To get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us based on what it says or does not say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Entersoft Australia and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Entersoft) owe no duty of care towards you or any other person, nor does Entersoft make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and Entersoft hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Entersoft hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Entersoft, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the Smart contract is purely based on the smart contract code shared with us alone.

2.0 Overview

2.1 Project Overview

During the period of **22 September 2025 to 22 October 2025**, Entersoft performed smart contract security audits for **DeFa**.

2.2 Scope

The scope of this audit encompassed a thorough analysis and documentation of the smart contract codebase, with a focus on three key areas: quality, security, and correctness. The primary objective was to ensure that the code not only adheres to industry best practices but also functions as intended without introducing vulnerabilities or logic flaws.

File In Scope:

- Defa-H.sol ,
- Defa-L.sol ,
- Defa-M.sol,
- MultiSigWallet.sol ,
- poolContract.sol ,
- PoolDeployer.sol ,
- Whitelist.sol

Commit id: 517d127c4c28a36e9790dcb986aaaf7127b3aa96

Out of Scope: External contracts, External Oracles, other smart contracts in the repository, or imported smart contracts.

2.3 Project Summary

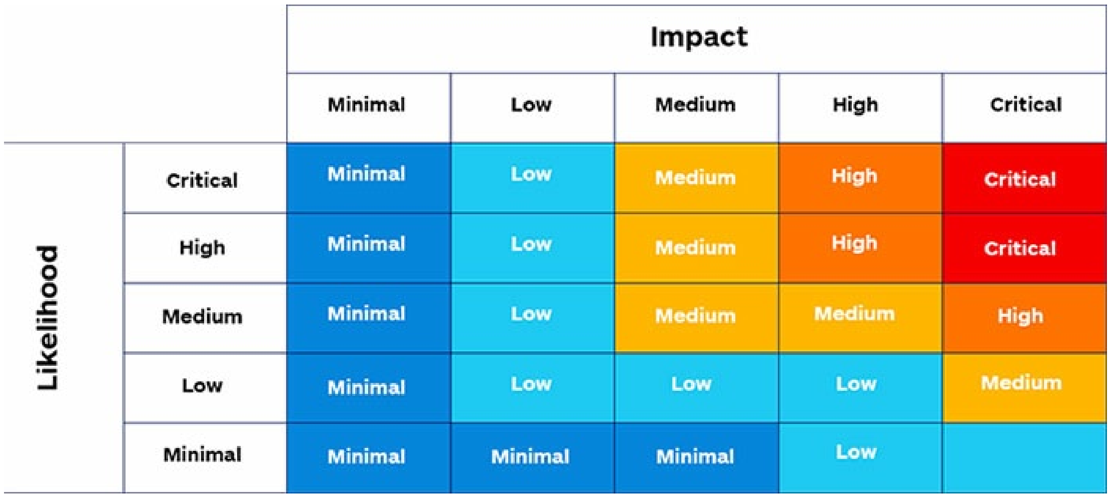
Project Name	No. of Smart Contract File(s)	Verified	Vulnerabilities
DeFa	7	Yes	As per report. Section 2.6

2.4 Audit Summary

Delivery Date	Method of Audit	Consultants Engaged
23 October 2025	Manual and Automated Approach	3

2.5 Security Level References

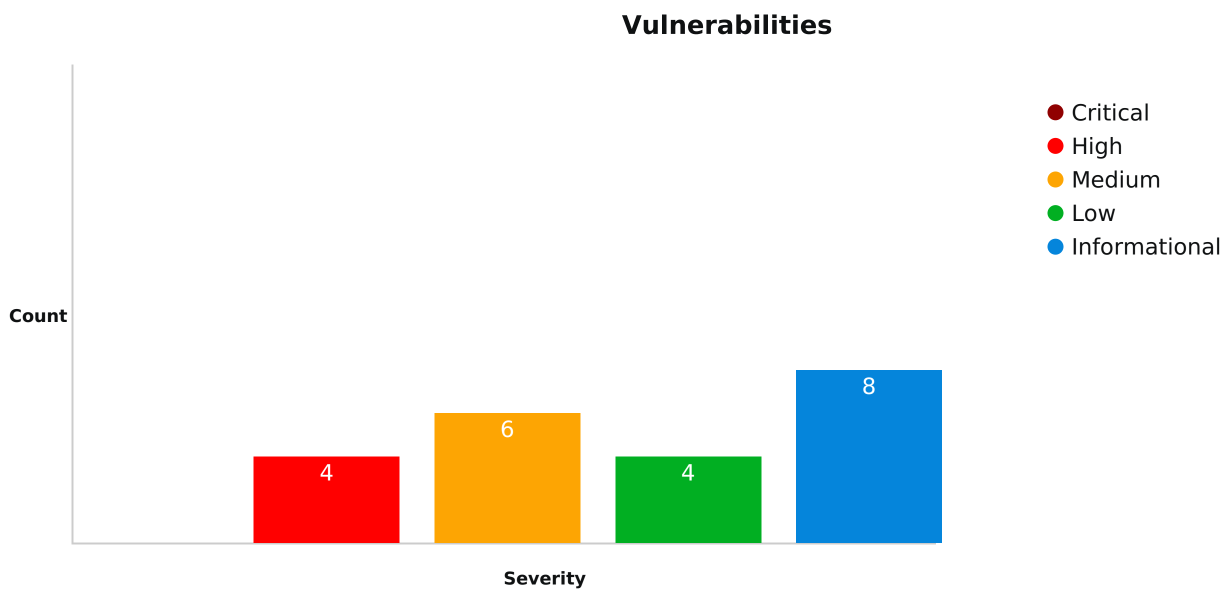
Every vulnerability in this report was assigned a severity level from the following classification table:



		Impact				
		Minimal	Low	Medium	High	Critical
Likelihood	Critical	Minimal	Low	Medium	High	Critical
	High	Minimal	Low	Medium	High	Critical
	Medium	Minimal	Low	Medium	Medium	High
	Low	Minimal	Low	Low	Low	Medium
	Minimal	Minimal	Minimal	Minimal	Low	

2.6 Vulnerability Summary

● Critical	● High	● Medium	● Low	● Informational
0	4	6	4	8



3.0 Executive Summary

Entersoft conducted a comprehensive Smart Contract Security Audit of the Defa Protocol, applying a multi-layered assessment framework that combined automated, manual, and adversarial analysis techniques.

The primary objective was to identify and mitigate potential vulnerabilities, logic flaws, and security risks within the protocol's Solidity smart contracts, ensuring alignment with industry best practices and the highest standards of security, reliability, and performance.

Our audit encompassed:
Manual code review to evaluate core logic, access controls, and edge-case handling.

Static and dynamic analysis using industry-standard tooling and custom scripts to detect vulnerabilities across the codebase.

Business logic and exploit simulation testing, including adversarial threat modelling to assess resilience against real-world attack vectors.

Remediation validation, confirming that identified issues were appropriately addressed and resolved.

The audit was performed between 13 October 2025 and 23 October 2025, during which Entersoft's security engineers thoroughly assessed and validated the Defa Protocol's security posture. The engagement resulted in the identification of several vulnerabilities, each categorised by severity and detailed within the Vulnerability Summary Table and associated technical sections of this report.

This assessment reflects Entersoft's commitment to delivering a rigorous, transparent, and assurance-driven review process providing stakeholders with confidence that the Defa Protocol has undergone a high-standard, comprehensive security evaluation.

Testing Methodology

Entersoft's audit of the Defa Protocol followed a rigorous, standards-aligned methodology that integrates both automated and manual assessment techniques, grounded in recognised smart contract security frameworks and global assurance benchmarks.

Our process commenced with extensive static and dynamic analysis using industry-leading tools such as Slither and Aderyn to automatically identify potential vulnerabilities across the Solidity codebase. Building upon this, our engineers performed a manual, line-by-line code review to uncover complex logic and business-layer vulnerabilities that automated scanners alone may overlook. This included targeted testing against high-risk exploit classes such as reentrancy, arithmetic overflow and underflow, access-control bypasses, and timestamp manipulation.

The audit further incorporated scenario-based testing and edge-case simulation, designed to validate the protocol's resilience under stressed operational conditions. Both positive and negative test cases were constructed to ensure comprehensive coverage and graceful error handling.

Our methodology aligns with Solidity Security Guidelines, the Smart Contract Security Verification Standard (SCSVS), and broader cybersecurity frameworks including OWASP Application Security Verification Standard (ASVS) and NIST SP 800-53 controls for cryptography, access management, and integrity assurance. Supplementary tools such as Solhint (linting and code quality), sol-profiler, sol-coverage, and sol-sec were employed to optimise code efficiency, measure test completeness, and eliminate redundant dependencies.

This standards-driven approach ensures a deep, systematic, and verifiable security evaluation — providing confidence that the protocol has been tested in accordance with leading global best practices for Solidity-based smart contracts.

Tools Used for Audit

During the audit we combined manual expertise with automated analysis to ensure a thorough review of the protocol's security and performance. The team used a range of tools within the audit process such as Slither and Aderyn to improve detection accuracy and assess code quality. This approach, built on industry best practices and careful review, helped identify potential vulnerabilities that automated tools might miss. Entersoft's process focuses on precision, depth and maintainability to ensure a secure and reliable smart contract codebase.

Code Review and Manual Analysis

A detailed manual review of the Solidity contracts was carried out to verify the results from automated tools and uncover any additional vulnerabilities. Each line of code was carefully reviewed to ensure it met the guidelines and requirements shared during the onboarding phase.

Through this hands-on analysis the audit team confirmed the implementation of security controls, checked logical accuracy and identified issues that static analysis tools might have missed. This combined approach helped ensure the smart contract is both secure and reliable.

Auditing Approach and Methodologies Applied

The assessment focused on the following key areas:

- **Code Quality and Structure:**
  The team reviewed the codebase in detail to identify issues affecting structure, readability, and maintainability. This included evaluating the architecture of the smart contract to confirm alignment with best coding practices and standards.

- **Security Vulnerabilities:**
  Manual analysis techniques were applied to identify security weaknesses that could be exploited by attackers. This included the detection of buffer overflows, injection risks, weak or deprecated cryptographic functions, and signature handling vulnerabilities.

Documentation Overview

Comprehensive documentation is a critical component of secure and maintainable protocol development. It ensures transparency, assists developers and auditors in understanding the system's architecture and functionality, and supports long-term maintenance and troubleshooting.

For blockchain-based protocols, detailed documentation is especially important due to the complexity and security requirements of decentralized systems. The following documentation artifacts were reviewed and rated based on their quality and completeness as provided by the protocol team:

Protocol Documentation:

- **Purpose**: To describe the overall protocol, its goals, architecture, and how it operates.
- **Quality Provided**: Good

Technical Documentation:

- **Purpose**: To provide detailed information about the technical aspects of the protocol, including smart contracts, APIs, and their interactions.
- **Quality Provided**: Good

Inline Comments:

- **Purpose**: To explain specific parts of the code, such as the purpose of functions, variables, and logic.
- **Quality Provided:** Good

Smart Contract Audit Testing objectives	Result
OVERALL SECURITY POSTURE	● **POOR**

3.1 Findings

Vulnerability ID	Contract Name	Severity	Status
1	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● High	Identified
2	Arbitrum-marketplace-Audit-beta/contracts/poolContract.sol	● High	Identified

3	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● High	Identified
4	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● High	Identified
5	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Medium	Identified
6	Arbitrum-marketplace-Audit-beta\contracts\PoolDeployer.sol	● Medium	Identified
7	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Medium	Identified
8	Arbitrum-marketplace-Audit-beta\contracts\MultiSigWallet.sol	● Medium	Identified
9	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Medium	Identified
10	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Medium	Identified
11	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Low	Identified
12	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol, Whitelist.sol	● Low	Identified
13	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Low	Identified
14	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Low	Identified
15	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Informational	Identified
16	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Informational	Identified
17	Defa-H, Defa-M, Defa-L	● Informational	Identified
18	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Informational	Identified
19	poolContract.sol	● Informational	Identified

20	poolContract.sol	● Informational	Identified
21	Arbitrum-marketplace-Audit-beta\contracts\Whitelist.sol	● Informational	Identified
22	Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol	● Informational	Identified

3.2 Recommendations

Entersoft's evaluation found that the Defa Protocol demonstrates a strong architectural foundation and alignment with its intended business model. However, several logic and security enhancements are required to reach full production-grade assurance.

Key findings included:

- Critical and High-Severity Issues: Vulnerabilities within participation logic impacting protocol integrity and an operational flaw in the emergency withdrawal function (ignores recipient parameter).
- Medium-Severity Findings: Input validation weaknesses, inconsistent event emissions, and insufficient administrative control checks.
- Low-Severity and Best-Practice Enhancements: Typos, naming inconsistencies, redundant initialisations, and compiler version misalignment.

Further, we identified opportunities to strengthen protocol resilience through:

- Enforcing strict access-control and pausing mechanisms.
- Implementing unit and fuzz tests for critical flows such as repayment ordering, reconstruction, and distribution fairness.
- Adopting OpenZeppelin SafeERC20 standards and non-reentrant modifiers for fund-moving functions.
- Standardising time unit declarations, tightening constructor parameters, and ensuring immutable protocol constants for improved predictability.

After applying the recommended remediations, the protocol's residual risk is assessed as manageable, with the system well-positioned for mainnet deployment. Addressing high-impact vulnerabilities first, followed by medium- and low-severity improvements, will ensure a hardened, maintainable, and regulator-ready implementation.

4.0 Technical Analysis

4.1 Blacklist Check Can Be Bypassed on Claim Functions

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

Claim functions in the `poolContract` (`claimLiquidity`, `claimInsurance`, and `claimLpToken)`restricts blacklisted lenders from claiming funds by checking `isBlacklisted[msg.sender]`. However, these functions allow any user to pass an arbitrary `_lender` address as a parameter. This means even if a lender is blacklisted, they can call the functions from another non-blacklisted address by passing the main _lender address and claim funds, effectively bypassing the blacklist enforced check.

Locations:

NA

Remediation:

Update all claim functions (`claimLiquidity`, `claimInsurance`, `claimLpToken`) to:

1. Check the blacklist status of the actual (`_lender`), not just `msg.sender`.
2. Restrict claims so only the lender themselves can claim (i.e., require `msg.sender == _lender`).

Code fix:

```
function claimLiquidity(address _lender) public whenNotPaused nonReentrant {
require(!isBlacklisted[_lender], "Blacklisted lender");
require(msg.sender == _lender, "Only lender can claim");
...
}
```

Impact:

- Blacklisted lenders can still receive funds by having a non-blacklisted address call the claim functions on their behalf.

Code Snippet:

```
        function claimLpToken(address _lender) public whenNotPaused nonReentrant {

require(!isBlacklisted[msg.sender], "Blacklisted lender");

//@audit: can be bypassed

require(isEnableClaimBack, "Wait for the repayments");

LenderDetails storage lender = _lenederDetails[_lender];

uint256 mintAbleToken = lender.claimAbleLpToken;

require(mintAbleToken > 0, "nothing to claim");

lpToken.mint(_lender, mintAbleToken);

lender.claimedLpToken = lender.claimedLpToken + (mintAbleToken);

lender.claimAbleLpToken = 0;

}
```

Reference:

NA

POC's Description:

NA

Proof of Vulnerability:

N.A.

4.2 Incorrect _poolValues length check in initialize

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta/contracts/poolContract.sol

Description:

In `initialize`, the new guard requires `len == _poolValues.length`. `_poolValues` is always the 4 element configuration array (`[matureMinutes, loanTenureMinutes, poolAPY%, imAPY%]`), so the check now insists every pool have exactly four invoices. Any attempt to deploy a pool with a different invoice count reverts with “values are not equivalent”, even though borrower/amount/min arrays are aligned.

Locations:

poolContract.sol

- line no. 133

Remediation:

To resolve the issue :

Remove the `_poolValues` comparison from the array-length require. Instead, validate only that `_invoicesIds`, `_amounts`, `_minAmounts`, and `_borrowerAddresses` share the same length. Separately assert that `_poolValues.length == 4` (or enforce via a constant) to keep the configuration contractually fixed while keeping flexibility in invoice counts.

Impact:

As a impact of the issue Pool creation breaks for core business scenarios where invoice batches aren’t exactly four entries. Admins can no longer launch pools sized to their borrower set, blocking capital formation and any downstream participation, repayment, or restructuring flows.

Code Snippet:

```
        require(
len == _amounts.length &&
len == _minAmounts.length &&
len == _borrowerAddresses.length &&
len == _poolValues.length,
"values are not equivalent"
);
```

Reference:

NA

POC's Description:

NA

Proof of Vulnerability:

N.A.

4.3 Missing Insurance Status Validation

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

Inside the loop, the function sets `payDetails.isInsured = true` without verifying whether the invoice is already insured. As a result, an admin could re-insure the same invoice multiple times, overwriting existing insurance details and potentially manipulating insured amounts.

Locations:

poolContract - 552

Remediation:

Add a validation check before updating in the for loop:

```
for (uint256 i = 0; i < _invoice.length; i++) {
InvoicePaymentDetails storage payDetails = _invoicePaymentDetails[_invoice[i]];
require(payDetails.isDefault, "not default");
require(!payDetails.isInsured,"Invoice already insured");
payDetails.isInsured = true;
payDetails.amountInsured = _amount[i]; amount = amount + (_amount[i]);
}
```

Impact:

- Insurance data integrity can be compromised.
- Previously insured invoices may have their values overwritten.
- It can lead to incorrect accounting of total insured amounts and possible fund misallocation.

Code Snippet:

```
        function fillInsurance(string[] memory _invoice, uint256[] memory _amount) public onlyAdminSafe {

uint256 amount;

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

InvoicePaymentDetails storage payDetails = _invoicePaymentDetails[_invoice[i]];

require(payDetails.isDefault, "not default");

payDetails.isInsured = true;

payDetails.amountInsured = _amount[i];

amount = amount + (_amount[i]);

}

require(amount <= principalToRepay, "amount should be less or eqvalent");

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

LenderDetails storage lender = _lenederDetails[lendersAddresses[i]];

if (!isBlacklisted[lendersAddresses[i]]) {

uint256 shareHolding = getShareHolding(lendersAddresses[i]);

uint256 claimAble = (shareHolding * (amount)) / (percentDivider);

lender.claimAbleInsurance = claimAble;

}

}

}
```

Reference:

Proof of Vulnerability:

N.A.

4.4 No Minimum Deposit on Participation

Severity	Status	Type of Analysis
● High	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

The `participate` function in `poolContract` allows any user to participate in the pool with any `_amount` value, including zero or extremely small amounts. There is no minimum deposit enforced. Each new participant address is added to the `lendersAddresses` array, regardless of the deposit size.

This design allows a malicious user to use multiple addresses (Sybil attack) to make many tiny deposits, bloating the `lendersAddresses` array. Since several protocol functions (such as `addingClaimables` and `fillInsurance`) iterate over this array, excessive growth can lead to high gas costs or even denial-of-service (DoS) if transactions run out of gas.

Locations:

NA

Remediation:

- **Enforce a Minimum Deposit:**
  Add a constant (e.g., `MIN_DEPOSIT`) and require `_amount >= MIN_DEPOSIT` in the `participate` function.

  code fix:

  ```
  // ...existing code...
  uint256 public constant MIN_DEPOSIT = 100 * 1e18; // Example: 100 USD

  function participate(uint256 _amount) public whenNotPaused {
      require(!isBlacklisted[msg.sender], "Blacklisted lender");
      require(
          whitelistAddress.isWhitelisted(msg.sender),
          "User not Whitelisted"
      );
      require(!poolExecuted, "Pool is already executed");
      require(_amount >= MIN_DEPOSIT, "Deposit too small");
      // ...existing code...
  ```

```
    }
```

Impact:

- Functions that iterate over `lendersAddresses` (e.g., distributing claimables, insurance payouts) may become too expensive or impossible to execute if the array is excessively large, potentially freezing core protocol operations.
- A single entity can create many lender entries with minimal cost, skewing any logic that depends on the number of lenders (e.g., governance, airdrops, or rewards).
- Unnecessary storage of many critical lender records increases contract state size and long-term costs.
- Without a minimum deposit, there is no economic disincentive to spamming the protocol.

Code Snippet:

```
        function participate(uint256 _amount) public whenNotPaused {

require(!isBlacklisted[msg.sender], "Blacklisted lender");

require(

whitelistAddress.isWhitelisted(msg.sender),

"User not Whitelisted"

);

require(!poolExecuted, "Pool is already executed");

LenderDetails storage lender = _lenederDetails[msg.sender];

if (!lender.isExists) {

lender.isExists = true;

lendersAddresses.push(msg.sender);

}

require(

_amount + (amountRaised) <= hardCap,

"Amount is exceeding the hardCap"

);

require(block.timestamp <= poolMatureTime, "Can't participate Now");

require(!isPoolMature, "Pool is mature already");

bool success = USD.transferFrom(msg.sender, address(this), _amount);

require(success, "Not Transferred");

lpToken.mint(msg.sender, _amount * (tokenMultiplier));

lender.poolShare = lender.poolShare + (_amount);

amountRaised = amountRaised + (_amount);

}
```

Reference:

NA

POC's Description:

NA

Proof of Vulnerability:

N.A.

4.5 Incorrect Recipient in emergencyWithdraw Function Allows Only Transfers to imReciver

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

The emergencyWithdraw function in the poolContract contract is designed to allow the treasurySafe address to withdraw ERC20 tokens from the contract in emergency situations. The function accepts three parameters: _token (the token address), _to (the intended recipient), and _amount (the amount to withdraw). However, the implementation ignores the _to parameter and always transfers tokens to the imReciver address. This means that regardless of the _to address provided, the tokens will always be sent to imReciver. The _to parameter is therefore misleading and ineffective.

Locations:

NA

Remediation:

If the intention is to allow withdrawal to any address:
Update the function to use the _to parameter as the recipient.

If the intention is to always send to imReciver:
Remove the _to parameter from the function signature and all calls, and update documentation to reflect this behavior.

Impact:

- The function cannot be used to send tokens to arbitrary addresses in emergencies, which may be required for recovery, migration, or other operational needs.
- The presence of the _to parameter suggests that the function can send tokens to any address, which is not the case. This could lead to confusion or operational errors.
- If an emergency requires sending funds to a specific address (e.g., a new treasury or recovery wallet), this function will not fulfill that requirement.

Code Snippet:

```
        function emergencyWithdraw(

address _token,

address _to,

uint256 _amount

) external onlyTreasurySafe {

require(_to != address(0), "Invalid Recipient");

IERC20(_token).transfer((imReciver), _amount);

}
```

Reference:

NA

Proof of Vulnerability:

N.A.

4.6 Incorrect usage of __Ownable_init() with parameters which is incompatible with OpenZeppelin v4.9

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\PoolDeployer.sol

Description:

The initialize() function incorrectly passes an argument to the __Ownable_init() function.
In OpenZeppelin v4.9, the OwnableUpgradeable contract's initializer function __Ownable_init() does not accept any parameters, unlike in OpenZeppelin v5.x, where ownership can be initialized with a custom owner address.

This mismatch causes a compilation error.

The protocol is using OpenZeppelin Contracts v4.9, where the initializer function __Ownable_init() is defined as:

```
function __Ownable_init() internal onlyInitializing {
        __Ownable_init_unchained();
}
```

It does not accept an address argument.

Passing _multisig as a parameter is only valid in OpenZeppelin v5.x, which introduced parameterized ownership initialization.

Locations:

- Arbitrum-marketplace-Audit-beta\contracts\PoolDeployer
    - Line Number: 38

Remediation:

Use the Correct OpenZeppelin version, which is a compatible pattern for ownership initialization.

Either the protocol could use the v5 version or implement the initialization correctly according to v4.9.

Impact:

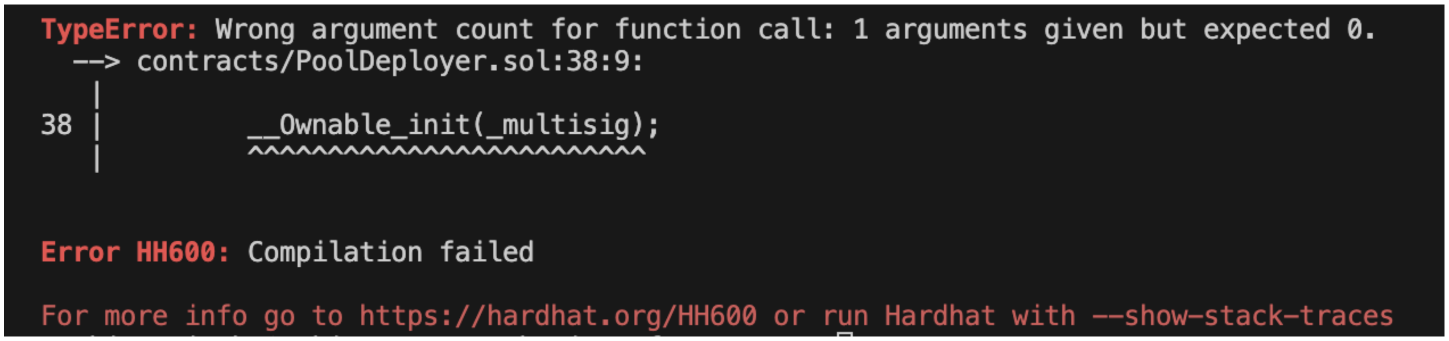
- Compilation fails, preventing deployment of the contract.
- The ownership of the contract will not be properly initialized until corrected.

Code Snippet:

```
        function initialize(address _multisig) external initializer {

__Ownable_init(_multisig);

__UUPSUpgradeable_init();

defaLow = 0x3Cb67031Cd7BD585f56215bF05AabF98D71D9EcC;

defaMedium = 0xc1dDA0180b2B60BD5d50360FBC0283CB058c6F19;

defaHigh = 0xa5256b5B799c9256F0f8a27aC4b1e4D6FD364f7D;

}
```

Reference:

Proof of Vulnerability:



```
TypeError: Wrong argument count for function call: 1 arguments given but expected 0.
  --> contracts/PoolDeployer.sol:38:9:
   |
38 |         __Ownable_init(_multisig);
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^


Error HH600: Compilation failed

For more info go to https://hardhat.org/HH600 or run Hardhat with --show-stack-traces
```

4.7 Lacks event emission for critical changes

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

The `poolContract` lacks event emission for several critical state changing functions, including participation, claims (liquidity, insurance, LP tokens), repayments, pool execution, and emergency actions. While a few events exist (e.g., `ChangeAdmin`, `UpadtePercentage`), they are not consistently emitted across all major contract actions.

Locations:

NA

Remediation:

Add event definitions and emit them in all critical functions.
Below are suggested events and where to emit them:

```
event Participated(address indexed lender, uint256 amount);
event ClaimedLiquidity(address indexed lender, uint256 amount);
event ClaimedInsurance(address indexed lender, uint256 amount);
event ClaimedLpToken(address indexed lender, uint256 amount);
event LoanRepaid(address indexed borrower, string invoiceId, uint256 amount);
event PoolExecuted(uint256 totalDisbursed, uint256 timestamp);
event EmergencyWithdraw(address indexed token, address indexed to, uint256 amount);
event PoolClosed(uint256 timestamp);
```

Emit those events in the respective functions.

Impact:

- Without events, off-chain systems (such as dApps, analytics dashboards, and monitoring systems) cannot reliably track or verify contract activity. This reduces transparency for users and stakeholders.
- Users and integrators rely on events to receive real time updates about their actions (e.g., successful participation, claim, or repayment). The absence of events can lead to confusion and bad UX.

- Events are essential for post deployment auditing, dispute resolution, and regulatory compliance. Missing events make it difficult to reconstruct the contract’s history and verify correct operation.
- Many DeFi protocols and monitoring tools depend on events to trigger automated processes (e.g., updating balances, sending notifications). Lack of events can break these integrations.

Code Snippet:

NA

Reference:

NA

Proof of Vulnerability:

N.A.

4.8 Missing Admin Functionality for Changing Confirmation Threshold in MultiSigWallet

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\MultiSigWallet.sol

Description:

The MultiSigWallet contract sets the number of required confirmations (NUMBER_OF_CONFIRMATIONS_REQUIRED) only once during deployment, via the constructor. There is **no function to change this threshold** after deployment. The contract does not allow the owners to increase or decrease the number of confirmations required for transaction execution which may be a needed for future changes.

Locations:

NA

Remediation:

Add an admin only function to change the confirmation threshold.
This function should:

- Only be callable by the wallet itself (i.e., via a multisig transaction approved by the current threshold), or by a designated admin if protocol governance allows.
- Ensure the new threshold is greater than zero and less than or equal to the number of owners.

code fix:

```
// ...existing code...

function changeConfirmationThreshold(uint256 newThreshold) public onlyOwner {
    require(
        newThreshold > 0 && newThreshold <= owners.length,
        "Invalid new threshold"
    );
    NUMBER_OF_CONFIRMATIONS_REQUIRED = newThreshold;
    // Optionally emit an event for off-chain tracking
    emit ConfirmationThresholdChanged(newThreshold);
}
```

```
// Add event at the top:
event ConfirmationThresholdChanged(uint256 newThreshold);

// ...existing code...
```

Impact:

The lack of a admin function to change the confirmation threshold after deployment limits the flexibility, security, and operational resilience of the `MultiSigWallet`

Code Snippet:

NA

Reference:

NA

Proof of Vulnerability:

N.A.

4.9 Missing Array Length Validation

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

The function `fillInsurance()` accepts two dynamic arrays: `_invoice` and `_amount`. However, there is no validation to ensure that both arrays have the same length. This can lead to out-of-bound reads or mismatched insurance allocations, where the insurance amount array doesn't correspond correctly to the invoice array.

Locations:

poolContract - 550

Remediation:

Add a validation check at the beginning of the function:

```
require(_invoice.length == _amount.length, "Array length mismatch");
```

Impact:

If the arrays differ in length, the loop may access invalid indices, leading to a **revert** or **incorrect insurance mapping**, breaking consistency in state updates.

Code Snippet:

```
        function fillInsurance(string[] memory _invoice, uint256[] memory _amount) public onlyAdminSafe {

uint256 amount;

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

InvoicePaymentDetails storage payDetails = _invoicePaymentDetails[_invoice[i]];

require(payDetails.isDefault, "not default");

payDetails.isInsured = true;

payDetails.amountInsured = _amount[i];

amount = amount + (_amount[i]);

}

require(amount <= principalToRepay, "amount should be less or eqvalent");

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

LenderDetails storage lender = _leneder Details[lendersAddresses[i]];

if (!isBlacklisted[lendersAddresses[i]]) {

uint256 shareHolding = getShareHolding(lendersAddresses[i]);

uint256 claimAble = (shareHolding * (amount)) / (percentDivider);

lender.claimAbleInsurance = claimAble;

}

}

}
```

Reference:

Proof of Vulnerability:

N.A.

4.10 Zero address check missing

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

The poolContract.sol contract does not validate whether the parameters _USD, defa, _addresses and _lpToken passed in `initialize` and `updateTokenAddresses` functions are non-zero addresses. Zero addresses are often used as placeholders or default values, and failing to properly validate them can result in unintended behavior.

Locations:

- Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol
    - Line Numbers: 134, 157, 159, 636

Remediation:

Add checks in the functions to ensure zero address is not passed in the initialize` and `updateTokenAddresses`.

Impact:

Deploying the contract with zero addresses for critical roles could lead to misconfiguration,rendering the contract unusable or insecure.

Code Snippet:

```
    function initialize(

address[] memory _addresses,

address _USD,

address defa,

string[] memory _invoicesIds,

address[] memory _borrowerAddresses,

uint256[] memory _amounts,

uint256[] memory _minAmounts,

uint256[4] memory _poolValues
```

```
) external returns (uint256 minLimit, uint256 maxLimit) {

uint256 _hardCap;

uint256 _softCap;

require(!initialized, "Already initialized");

require(

_borrowerAddresses.length == _amounts.length && _borrowerAddresses.length == _minAmounts.length,

"values are not equivalent"

);

lpToken = IERC20(defa);

launchTime = block.timestamp;

poolMatureTime = block.timestamp + (_poolValues[0] * (1 minutes));

bufferTime = poolMatureTime + 1 minutes;

poolDuration = _poolValues[1];

uint256 endTime = poolEndTime = bufferTime + (poolDuration * (1 minutes));

poolApy = _poolValues[2] * (1e10);

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

_invoiceDetails[_invoicesIds[i]].principalAmount =

_invoiceDetails[_invoicesIds[i]].principalAmount + (_amounts[i]);

_invoiceDetails[_invoicesIds[i]].borrower = _borrowerAddresses[i];

_invoiceDetails[_invoicesIds[i]].isExists = true;

_invoiceDetails[_invoicesIds[i]].minAmount = _invoiceDetails[_invoicesIds[i]].minAmount + (_minAmounts[i]);

_invoicePaymentDetails[_invoicesIds[i]].endTime = endTime;

_softCap += (_minAmounts[i]);

_hardCap += (_amounts[i]);

}

hardCap = _hardCap;

softCap = _softCap;

minLimit = softCap;

maxLimit = hardCap;
```

```
admin = payable(_addresses[0]);

treasurySafe = payable(_addresses[1]);

whitelistAddress = IWhitelist(_addresses[2]);

initialized = true;

USD = IERC20(_USD);

imApy = _poolValues[3] * (1e10);

markup = imApy + (poolApy);

}






function updateTokenAddresses(address _lpToken) external onlyAdminSafe {

require(!poolExecuted, "Already Executed");

lpToken = IERC20(_lpToken);

}
```

Reference:

Proof of Vulnerability:

N.A.

4.11 changeAdmin Function Lacks Two-Step Process and Emit Event

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

The `changeAdmin` function in the `poolContract` allows the current admin to immediately transfer administrative control to a new address in a single transaction.

The function does not require confirmation from the new admin address. If the current admin mistakenly enters an incorrect address, or if the transaction is compromised, control of the contract could be irreversibly lost or transferred to an unintended party.

Although the `ChangeAdmin` event is defined in the contract, it is not emitted when the admin is changed.

Locations:

NA

Remediation:

Implement a Two-Step Admin Change Process:

- **Step 1: Propose New Admin**

```
address public pendingAdmin;

event AdminProposed(address indexed newAdmin);

function proposeAdmin(address _newAdmin) external onlyAdminSafe {
    require(_newAdmin != address(0), "admin=0");
    pendingAdmin = _newAdmin;
    emit AdminProposed(_newAdmin);
}
```

Step 2: Accept Admin Role

```
function acceptAdmin() external {
    require(msg.sender == pendingAdmin, "Not pending admin");
    admin = payable(pendingAdmin);
    pendingAdmin = address(0);
    emit ChangeAdmin(admin);
}
```

- Emit the `ChangeAdmin` Event in the Current Function

Impact:

If the admin address is set to an incorrect or inaccessible address, the contract could become unmanageable, with no way to recover administrative control.

Code Snippet:

```
    function changeAdmin(address _newAdmin) external onlyAdminSafe {

  require(_newAdmin != address(0), "admin=0");

  admin = payable(_newAdmin);

}
```

Reference:

NA

Proof of Vulnerability:

N.A.

4.12 State variables that could be declared immutable

Severity	Status	Type of Analysis
● Low	Identified	Static

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol, Whitelist.sol

Description:

In Solidity, certain state variables are initialized once in the constructor and never modified afterwards. If such variables are not explicitly declared as immutable, they remain stored in contract storage rather than in bytecode. This leads to unnecessary gas costs for every read operation since storage access is more expensive than reading an immutable variable from bytecode.

State variables that are not updated following deployment should be declared immutable to save gas.

Locations:

- Arbitrum-marketplace-Audit-beta\contracts\poolContractsol
  - Line Numbers: 111, 112
- Arbitrum-marketplace-Audit-beta\contracts\Whitelist.sol
  - Line Number: 5

Remediation:

Add the immutable attribute to state variables that never change or are set only in the constructor.

poolContract.sol

```
uint256 public immutable percentDivider = 0;
uint256 public immutable tokenMultiplier = 0;

Whitelist.sol

address public immutable owner;
```

Impact:

While this issue doesn't directly impact the functionality of the contract, the contract can benefit from the use of constant and immutable keywords for variables that do not change after deployment. This can save gas costs by storing the variables directly in the bytecode.

Code Snippet:

```
        Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol


uint256 public percentDivider = 0;

uint256 public tokenMultiplier = 0;



Arbitrum-marketplace-Audit-beta\contracts\Whitelist.sol



address public owner;
```

Reference:

Proof of Vulnerability:

N.A.

4.13 Unchecked Return Value from ERC20 transfer() Function

Severity	Status	Type of Analysis
● Low	Identified	Static

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

The contract performs ERC20 token transfers using transfer() but**does not check the returned boolean value**.

According to the ERC20 standard, the transfer() function returns a bool indicating whether the operation succeeded or not.

Locations:

- Arbitrum-marketplace-Audit-beta\contracts\poolContract
    - Line Numbers: 386, 391, 647

Remediation:

Always **verify the success** of ERC20 transfers by checking the return value or by using**safe wrappers** from OpenZeppelin's SafeERC20 library, which properly handles ERC20 implementations.

Example:
IERC20(USD).safeTransfer(imReciver, paidIMYield);
IERC20(_token).safeTransfer(imReciver, _amount);

This ensures that:

- Non-reverting failures are caught.
- Tokens following both **strict ERC20** and **non-standard** implementations are handled safely.

Impact:

- If a token does not revert but returns false, the transfer may**silently fail**.
- The contract would **assume payment succeeded**, while tokens remain in the contract.
- This could cause **Accounting mismatches** (e.g. rewards or yields marked marked as paid but not actually sent).

Code Snippet:

```
USD.transfer(imReciver, paidIMYield);

IERC20(_token).transfer((imReciver), _amount);
```

Reference:

Proof of Vulnerability:

N.A.

4.14 Use external Visibility for Public Functions

Severity	Status	Type of Analysis
● Low	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

Within the poolContract.sol, certain functions are declared with the public visibility modifier but are never invoked internally by other contract functions. In Solidity, functions that are intended to be accessed only from outside the contract should instead be declared as external. Using public unnecessarily increases the compiled bytecode size and gas costs, since the compiler generates additional code to allow internal calls.

Locations:

NA

Remediation:

Change the visibility of functions that are not called internally from`public` to `external.`

Impact:

- `external` functions are slightly more gas efficient than `public` functions when called from outside the contract, as calldata is read directly rather than being copied to memory.
- Using `external` signals to developers and auditors that the function is not intended for internal use, improving readability and maintainability.
- This is not a security issue, but a best practice for contract design and optimization.

Code Snippet:

NA

Reference:

Proof of Vulnerability:

N.A.

4.15 nonReentrant Modifier is Not the First Modifier

Severity	Status	Type of Analysis
● Informational	Identified	Static

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

The functions use multiple modifiers, but the nonReentrant modifier is not placed first. Best practices recommend placing nonReentrant as the first modifier to ensure it executes before other modifiers.

Locations:

- Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol
  - Line Numbers: 255, 305

Remediation:

Reorder modifiers so that nonReentrant is always placed first.

```
function claimLiquidity(address _lender) public nonReentrant whenNotPaused {
```

```
function claimLpToken(address _lender) public nonReentrant whenNotPaused {
```

Impact:

This issue does not directly impact the protocol’s functionality. However, following this defensive coding practice reduces the attack surface if future modifiers are updated to include external calls.

Code Snippet:

```
function claimLiquidity(address _lender) public whenNotPaused nonReentrant {



function claimLpToken(address _lender) public whenNotPaused nonReentrant {
```

Reference:

Proof of Vulnerability:

N.A.

4.16 Default Initialization of State Variables

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

In Solidity, all state variables are automatically initialized to their default values when a contract is deployed. Specifically:

- All `uint256` (and other integer types) are initialized to `0`.
- All `bool` variables are initialized to `false`.
- All `address` variables are initialized to `address(0)`.

This means that explicitly assigning `= 0` to `uint256` or `= false` to `bool` variables in the contract is redundant and not required for correct behavior.

Locations:

NA

Remediation:

- You may safely remove explicit `= 0` and `= false` initializations from state variable declarations unless you want to highlight the default for documentation purposes.
- This will make the code cleaner and easier to read, with no change in behavior.

Impact:

- **No functional or security risk** is introduced by explicitly setting these values, but it can make the code unnecessarily verbose.
- **Best practice** is to rely on Solidity's default initialization for clarity and conciseness.

Code Snippet:

```
        // Pool configuration parameters

uint256 public hardCap = 0;

uint256 public softCap = 0;

uint256 public launchTime = 0;

uint256 public poolMatureTime = 0;

uint256 public bufferTime = 0;

uint256 public poolDuration = 0;

uint256 public poolEndTime = 0;

uint256 public totalAmountRepaid = 0;

uint256 public apyToRepay = 0;

uint256 public repaidApy = 0;

uint256 public IMApyToRepay = 0;

uint256 public repaidIMApy = 0;

uint256 public principalToRepay = 0;

uint256 public repaidPrincipal = 0;

uint256 public amountRaised = 0;

uint256 public poolApy = 0;

uint256 public imApy = 0;

uint256 public markup = 0;

uint256 public percentDivider = 0;

uint256 public tokenMultiplier = 0;

uint256 public reConstrcutionCount = 0;



// Pool status flags

bool public isEnableClaimBack = false;

bool public isPoolMature = false;

bool public initialized = false;

bool public poolClosed = false;

bool public poolExecuted = false;
```

Reference:

NA

Proof of Vulnerability:

N.A.

4.17 Flattened OZ library used in LP token contracts

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

Defa-H, Defa-M, Defa-L

Description:

LP token contracts embed the full OpenZeppelin library bodies instead of importing them. This indicates a flattened verification artifact was checked into source control rather than keeping the modular files.

Locations:

NA

Remediation:

Restore each LP contract to the standard structure (brief file with import statements referencing the appropriate OpenZeppelin packages). Ensure flattened output is generated only for on chain verification.

Impact:

- Security updates or bug fixes in OpenZeppelin can’t be pulled in via dependency bumps; every change would require manual edits in the flattened code.
- Breaks audit and build workflows that expect standard imports, complicates license checks, and risks divergence between audited and deployed bytecode.

Code Snippet:

NA

Reference:

NA

Proof of Vulnerability:

N.A.

4.18 Lack of Descriptive Error Messages in require Statement

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

In the `executePool` function, the `require` statements do not include descriptive error messages. When error messages are omitted, it becomes difficult for users, integrators, and developers to determine the exact cause of a transaction failure. Providing clear revert reasons is a recommended best practice, as it enables immediate feedback and facilitates debugging, especially in complex DeFi contracts where multiple conditions may lead to a revert.

Locations:

NA

Remediation:

- **Add clear, descriptive error messages** to all `require` statements, especially those validating user input or critical logic.

code fix:

```
require(
    amounts[i] >= _invoiceDetails[invoicesids[i]].minAmount &&
    amounts[i] <= _invoiceDetails[invoicesids[i]].principalAmount,
    "Amount must be within invoice min and max bounds"
);
```

Impact:

Without error messages, users and developers must manually inspect the code or use trial and error to determine why a transaction reverted.

Code Snippet:

```
        require(
     amounts[i] >= _invoiceDetails[invoicesids[i]].minAmount &&
     amounts[i] <= _invoiceDetails[invoicesids[i]].principalAmount
 );
```

Reference:

NA

Proof of Vulnerability:

N.A.

4.19 Lack of On-Chain Insuranace Transfer Logic in fillInsurance() Function

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

poolContract.sol

Description:

The fillInsurance() function is intended to fund insurance amounts from the admin/treasury wallet and credit them to the pool contract for lender protection in case of borrower default.
However, the function does not perform any actual on-chain transfer of tokens (e.g., via USD.transferFrom() or USD.transfer()).
Instead, the insurance amount is merely recorded in storage as:
payDetails.amountInsured = _amount[i];

This means the insurance fund accounting occurs only at the logical level within the contract, without verifiable token movement on-chain.

The Protocol is currently transferring funds manually from the admin wallet to the pool, and the contract simply fetches the available balance from the pool.
While functional in practice, this pattern creates a reliance on off-chain actions and could lead to inconsistencies between the recorded insurance amounts and the actual token balances in the pool.

Locations:

poolContract.sol:637-664

Remediation:

Consider integrating the insurance funding process directly within the fillInsurance() function to ensure that state updates correspond to actual token movements.

Impact:

- The contract state can reflect insured amounts even if the corresponding tokens were never transferred.
- The current setup is somewhat unconventional and could be streamlined by integrating direct transfer logic within the contract.

Code Snippet:

```
function fillInsurance(string[] memory _invoice, uint256[] memory _amount) public onlyAdminSafe {
    uint256 amount;
    for (uint256 i = 0; i < _invoice.length; i++) {
        InvoicePaymentDetails storage payDetails = _invoicePaymentDetails[_invoice[i]];
        require(payDetails.isDefault, "not default");
        payDetails.isInsured = true;
        payDetails.amountInsured = _amount[i];
        amount = amount + (_amount[i]);
    }
    require(amount <= principalToRepay, "amount should be less or eqvalent");


    for (uint256 i = 0; i < lendersAddresses.length; i++) {
        LenderDetails storage lender = _lenederDetails[lendersAddresses[i]];
        if (!isBlacklisted[lendersAddresses[i]]) {
            uint256 shareHolding = getShareHolding(lendersAddresses[i]);
            uint256 claimAble = (shareHolding * (amount)) / (percentDivider);
            lender.claimAbleInsurance = claimAble;
        }
    }
}
```

Reference:

Proof of Vulnerability:

N.A.

4.20 Potential Division by Zero in getShareHolding()

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

poolContract.sol

Description:

The function getShareHolding() computes a lender's shareholding percentage using the formula:
"shareHolding = (lender.poolShare * percentDivider) / amountRaised;"
However, the function does not validate whether amountRaised is greater than zero.

If this function is called before any lender participates in the pool, amountRaised will be 0, causing a division by zero revert.

Although this scenario may not occur under normal operational flow (as the function is expected to be called after funds are raised), it represents an unhandled edge case that could lead to unexpected reverts during early pool initialization or off-chain calls for analytics.

Locations:

poolContract.sol:677-683

Remediation:

Add a simple validation check to handle the zero state.

Impact:

- The function will revert if called before any deposits occur.
- No direct financial impact, but reduces robustness and clarity of contract behaviour.

Code Snippet:

```
function getShareHolding(address _lender) public view returns (uint256 shareHolding) {
    LenderDetails memory lender = _lenederDetails[_lender];
    shareHolding = (lender.poolShare * (percentDivider)) / amountRaised;  // ⚠ Division by zero
}
```

Reference:

Proof of Vulnerability:

N.A.

4.21 Solidity Pragma Should Be Specific, Not Wide and consistent through out the protocol

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\Whitelist.sol

Description:

In the smart contract, the pragma directive pragma solidity ^0.8.20; is used to specify the compiler version. This directive allows

any compiler version greater than or equal to 0.8.20;.

Locations:

- Arbitrum-marketplace-Audit-beta\contracts\Whitelist.sol
    - Line Number: 2
- Arbitrum-marketplace-Audit-beta\contracts\Interfaces\IWhitelist.sol
    - Line Number: 2

Remediation:

Update the pragma statements in the contracts to specify a particular version of Solidity. For example, replace pragma solidity

^0.8.20; with pragma solidity 0.8.20;.

And pragma version should be same across all the smart contracts, so change it to what other smart contracts in the project are using which is 0.8.29.

Impact:

This directive allows any compiler version greater than or equal to 0.8.20;. Failure to specify a specific version may lead to compatibility issues or unexpected behaviour in future compiler versions. It's important to follow best practices to ensure the stability and security of the contracts.

Code Snippet:

```
pragma solidity ^0.8.20;
```

Reference:

Proof of Vulnerability:

N.A.

4.22 Spelling/ Typos in PoolContract

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

Arbitrum-marketplace-Audit-beta\contracts\poolContract.sol

Description:

PoolContract code has several spelling and typo errors in variable names, function names, event names, and comments. While these do not directly affect contract execution, they can reduce code readability, increase the developer confusion, and may cause integration issues with off-chain systems or user interfaces.

Locations:

NA

Remediation:

- **Rename variables, functions, and events** to use correct spelling and consistent naming conventions (e.g., camelCase for variables/functions, PascalCase for events).
- **Update comments and annotations** for clarity and correctness.
- **Refactor usages throughout the contract** to match corrected names.

`_lenederDetails => _lenderDetails`

`reConstrcutionCount => reconstructionCount`

`UpadtePercentage => UpdatePercentage`

`fillIndivisualBuskets => fillIndividualBuckets`

Impact:

- Typos make the code harder to read and maintain.
- Off-chain tools, scripts, and UIs may misinterpret or fail to interact with events or variables due to inconsistent naming.
- Spelling errors reduce code quality during reviews.

Code Snippet:

NA

Reference:

NA

Proof of Vulnerability:

N.A.

5.0 Auditing Approach and Methodologies applied

Throughout the audit of the smart contract, care was taken to ensure:

- Overall quality of code
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Mathematical calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of token standards.
- Efficient use of gas.
- Code is safe from Re-entrancy and other vulnerabilities.

A combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

5.1 Structural Analysis

In this step we have analysed the design patterns and structure of all smart contracts. A thorough check was completed to ensure all Smart contracts are structured in a way that will not result in future problems.

5.2 Static Analysis

Static Analysis of smart contracts was undertaken to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

5.3 Code Review / Manual Analysis

Manual Analysis or review of done to identify new vulnerabilities or to verify the vulnerabilities found during the Static Analysis. The contracts were completely manually analysed, and their logic was checked and compared with the one described in the whitepaper. It should also be noted that the results of the automated analysis were verified manually.

5.4 Gas Consumption

In this step, we checked the behaviour of all smart contracts in production. Checks were completed to understand how much gas gets consumed, along with the possibilities of optimisation of code to reduce gas consumption.

5.5 Tools & Platforms Used For Audit

Slither, Aderyn

5.6 Checked Vulnerabilities

We have scanned DeFa smart contracts for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC-20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

6.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of DeFa and methods for exploiting them. Entersoft recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while Entersoft considers the major security vulnerabilities of the analyzed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the Smart Contract described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities will also change. Entersoft makes no undertaking to supplement or update this report based on changed circumstances or facts of which Entersoft becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that Entersoft use certain software or hardware products manufactured or maintained by other vendors. Entersoft bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, Entersoft does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by Entersoft for the exclusive benefit of DeFa and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between Entersoft and DeFa governs the disclosure of this report to all other parties including product vendors and suppliers.