CredShields
Smart Contract Audit

Mar 24th, 2023 • CONFIDENTIAL

Description

This document details the process and result of the JayContracts audit performed by CredShields Technologies PTE. LTD. on behalf of JayPeggers between Feb 26th, 2023, and March 10th, 2023. And a retest was performed on March 23rd, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

JayPeggers

Table of Contents

1. Executive Summary

JayPeggers engaged CredShields to perform a smart contract audit from Feb 26th, 2023, to March 10th, 2023. During this timeframe, Thirteen (13) vulnerabilities were identified. **A retest was performed on March 23rd, 2023, and all the bugs have been addressed.**

During the audit, Zero (0) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "JayPeggers" and should be prioritized for remediation, and fortunately, none were found.

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

Assets in Scope	Critical	High	Medium	Low	info	Gas	Σ
JayContracts	0	0	2	5	4	2	**13**
	0	**0**	**2**	**5**	**4**	**2**	**13**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in JayContracts's scope during the testing window while abiding by the policies set forth by JayContracts's team.

State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both JayPeggers's internal security and development teams to not only identify specific vulnerabilities, but also gain a deeper understanding of the current security threat landscape.

To ensure that vulnerabilities are not introduced when new features are added, or code is refactored, we recommend conducting regular security assessments. Additionally, by analyzing the root cause of resolved vulnerabilities, the internal teams at JayPeggers can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, JayPeggers can future-proof its security posture and protect its assets.

2. Methodology

JayPeggers engaged CredShields to perform a JayPeggers Smart Contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation phase

The CredShields team meticulously reviewed all provided documents and comments in the smart-contract code to gain a thorough understanding of the contract's features and functionalities. They meticulously examined all functions and created a mind map to systematically identify potential security vulnerabilities, prioritizing those that were more critical and business-sensitive for the refactored code. To confirm their findings, the team deployed a self-hosted version of the smart contract and performed verifications and validations during the audit phase.

A testing window from Feb 26th, 2023, to March 10th, 2023, was agreed upon during the preparation phase.

2.1.1 Scope

During the preparation phase, the following scope for the engagement was agreed-upon:

IN SCOPE ASSETS
https://github.com/toshimon-io/jay-contracts/tree/master/contracts

Table: List of Files in Scope

2.1.2 Documentation

Documentations was provided to the team as below.

https://jaypeggers.gitbook.io/whitepaper/

2.1.3 Audit Goals

CredShields uses both in-house tools (http://solidityscan.com/) and manual methods for comprehensive smart contract security auditing. The majority of the audit is done by manually reviewing the contract source code, following SWC registry standards, and an extended industry standard self-developed checklist. The team places emphasis on understanding core concepts, preparing test cases, and evaluating business logic for potential vulnerabilities.

2.2 Retesting phase

JayPeggers is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

CredShields follow OWASP's Risk Rating Methodology to determine the risk associated with discovered vulnerabilities. This approach considers two factors - Likelihood and Impact - which are evaluated with three possible values - **Low**, **Medium**, and **High**, based on factors such as Threat agents, Vulnerability factors, and Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

Overall, the categories can be defined as described below -

1. **Informational**

   We prioritize technical excellence and pay attention to detail in our coding practices. Our guidelines, standards, and best practices help ensure software stability and reliability. Informational vulnerabilities are opportunities for improvement and do not pose a direct risk to the contract. Code maintainers should use their own judgment on whether to address them.

2. **Low**

   Low-risk vulnerabilities are those that either have a small impact or can't be exploited repeatedly or those the client considers insignificant based on their specific business circumstances.

3. **Medium**

   Medium-severity vulnerabilities are those caused by weak or flawed logic in the code and can lead to exfiltration or modification of private user information. These vulnerabilities can harm the client's reputation under certain conditions and should be fixed within a specified timeframe.

4. **High**

   High-severity vulnerabilities pose a significant risk to the Smart Contract and the organization. They can result in the loss of funds for some users, may or may not require specific conditions, and are more complex to exploit. These vulnerabilities can harm the client's reputation and should be fixed immediately.

5. **Critical**

   Critical issues are directly exploitable bugs or security vulnerabilities that do not require specific conditions. They often result in the loss of funds and Ether from Smart Contracts or users and put sensitive user information at risk of compromise or modification. The client's reputation and financial stability will be severely impacted if these issues are not addressed immediately.

6. **Gas**

   To address the risk and volatility of smart contracts and the use of gas as a method of payment, CredShields has introduced a "Gas" severity category. This category deals with optimizing code and refactoring to conserve gas.

2.4 CredShields staff

The following individual at CredShields managed this engagement and produced this report:

- **Shashank, Co-founder CredShields**
    - shashank@CredShields.com

Please feel free to contact this individual with any questions or concerns you have around the engagement or this document.

3. Findings

This chapter contains the results of the security assessment. Findings are sorted by their severity and grouped by the asset and SWC classification. Each asset section will include a summary. The table in the executive summary contains the total number of identified security vulnerabilities per asset per risk indication.

3.1 Findings Overview

3.1.1 Vulnerability Summary

During the security assessment, Thirteen (13) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Floating Pragma	Low	Floating Pragma (SWC-103)
Hardcoded Static Address	Informational	Missing Best Practises
Use of SafeMath	Gas	Gas Optimization
Dead Code	Informative	Code With No Effects - SWC-135
Price Manipulation by Forcing Ether in the Contract	Medium	Business logic
Missing Events in important functions	Low	Missing Best Practices
Missing Multiple Zero Address Validations	Low	Missing Input Validation

Functions should be declared External	Gas	Gas Optimization
Unnecessary Multiple Payable Functions	Low	Missing Best Practices
Missing Price Feed Validation	Medium	Input Validation
Misspelled Variable/Typo	Informational	Missing best practices
Missing Input Validation in buyJay and buyNFTs	Low	Missing Input Validation
Incorrect Documentation	Informational	Insufficient Technical Documentation (CWE-1059)

Table: Findings in Smart Contracts

3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	Function Default Visibility	Not Vulnerable	Not applicable after v0.5.X (Currently using solidity v >= 0.8.6)
SWC-101	Integer Overflow and Underflow	Not Vulnerable	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Not Vulnerable	Version 0^.8.0 and above is used
SWC-103	Floating Pragma	Not Vulnerable	Contract uses floating pragma
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used
SWC-105	Unprotected Ether Withdrawal	Not Vulnerable	Appropriate function modifiers and require validations are used on sensitive functions that allow token or ether withdrawal.
SWC-106	Unprotected SELFDESTRUCT Instruction	Not Vulnerable	selfdestruct() is not used anywhere
SWC-107	Reentrancy	Not Vulnerable	No notable functions were vulnerable to it.
SWC-108	State Variable Default Visibility	Not Vulnerable	Not Vulnerable
SWC-109	Uninitialized Storage Pointer	Not Vulnerable	Not vulnerable after compiler version, v0.5.0

SWC-110	Assert Violation	Not Vulnerable	Asserts are not in use.
SWC-111	Use of Deprecated Solidity Functions	Not Vulnerable	None of the deprecated functions like block.blockhash(), msg.gas, throw, sha3(), callcode(), suicide() are in use
SWC-112	Delegatecall to Untrusted Callee	Not Vulnerable	Not Vulnerable.
SWC-113	DoS with Failed Call	Not Vulnerable	No such function was found.
SWC-114	Transaction Order Dependence	Not Vulnerable	Not Vulnerable.
SWC-115	Authorization through tx.origin	Not Vulnerable	tx.origin is not used anywhere in the code
SWC-116	Block values as a proxy for time	Not Vulnerable	Block.timestamp is not used
SWC-117	Signature Malleability	Not Vulnerable	Not used anywhere
SWC-118	Incorrect Constructor Name	Not Vulnerable	All the constructors are created using the constructor keyword rather than functions.
SWC-119	Shadowing State Variables	Not Vulnerable	Not applicable as this won't work during compile time after version 0.6.0
SWC-120	Weak Sources of Randomness from Chain Attributes	Not Vulnerable	Random generators are not used.
SWC-121	Missing Protection against Signature Replay Attacks	Not Vulnerable	No such scenario was found

SWC-122	Lack of Proper Signature Verification	Not Vulnerable	Not used anywhere
SWC-123	Requirement Violation	Not Vulnerable	Not vulnerable
SWC-124	Write to Arbitrary Storage Location	Not Vulnerable	No such scenario was found
SWC-125	Incorrect Inheritance Order	Not Vulnerable	No such scenario was found
SWC-126	Insufficient Gas Griefing	Not Vulnerable	No such scenario was found
SWC-127	Arbitrary Jump with Function Type Variable	Not Vulnerable	Jump is not used.
SWC-128	DoS With Block Gas Limit	Not Vulnerable	Not Vulnerable.
SWC-129	Typographical Error	Not Vulnerable	No such scenario was found
SWC-130	Right-To-Left-Override control character (U+202E)	Not Vulnerable	No such scenario was found
SWC-131	Presence of unused variables	Not Vulnerable	No such scenario was found
SWC-132	Unexpected Ether balance	Not Vulnerable	No such scenario was found
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Not Vulnerable	abi.encodePacked() or other functions are not used.
SWC-134	Message call with hardcoded gas amount	Not Vulnerable	Not used anywhere in the code
SWC-135	Code With No Effects	Not Vulnerable	No such scenario was found
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

4. Remediation Status

JayPeggers is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on March 23rd, 2023, and all the issues have been addressed.**

Also, the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Floating Pragma	Low	**Fixed [23/03/2023]**
Hardcoded Static Address	Informational	**Fixed [23/03/2023]**
Use of SafeMath	Gas	**Fixed [23/03/2023]**
Dead Code	Informative	**Fixed [23/03/2023]**
Price Manipulation by Forcing Ether in the Contract	Medium	**Invalid Bug [23/03/2023]**
Missing Events in important functions	Low	**Fixed [23/03/2023]**
Missing Multiple Zero Address Validations	Low	**Fixed [23/03/2023]**
Functions should be declared External	Gas	**Fixed [23/03/2023]**

Unnecessary Multiple Payable Functions	Low	**Fixed [23/03/2023]**
Missing Price Feed Validation	Medium	**Fixed [23/03/2023]**
Misspelled Variable/Typo	Informational	**Fixed [23/03/2023]**
Missing Input Validation in buyJay and buyNFTs	Low	**Fixed [23/03/2023]**
Incorrect Documentation	Informational	**Fixed [23/03/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Fixed]

Floating Pragma

Vulnerability Type
Floating Pragma (SWC-103)

Severity
Low

Description
Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities.
The contracts found in the repository were allowing floating or unlocked pragma to be used, i.e., **^0.8.17.**
This allows the contracts to be compiled with all the solidity compiler versions above the limit specified. The following contracts were found to be affected -

Affected Code

- contract JayMart
- contract JayLiquidityStaking
- contract JayFeeSplitter
- contract JAY

Impacts
If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions.

Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic.
The likelihood of exploitation is really low therefore this is only informational.

Remediation

Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use 0.8.16 pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest:

The bug has been fixed now a strict pragma is in use

Bug ID#2 [Fixed]

Hardcoded Static Address

Vulnerability Type

Missing Best Practises

Severity

Informational

Description

The contracts were found to be using hardcoded addresses.
This could have been optimized using dynamic address update techniques along with proper access control to aid in address upgrade at a later stage.

Affected Code

- contract JAY - https://github.com/toshimon-io/jay-contracts/blob/2e1c7255732d4dd5c3ef43824e7c40f53d789b3f/contracts/JayERC20.sol#L14
- contract JayMart - Line 34
- contract JayFeeSplitter - Line 13-18

```
  address payable private FEE_ADDRESS =
    payable(0x985B6B9064212091B4b325F68746B77262801BcB);


address payable private TEAM_WALLET =
    payable(0x985B6B9064212091B4b325F68746B77262801BcB);
address payable private LP_WALLET =
    payable(0x985B6B9064212091B4b325F68746B77262801BcB);
address payable private NFT_WALLET =
    payable(0x985B6B9064212091B4b325F68746B77262801BcB);
```

```
...

  address payable private constant TEAM_WALLET =
      payable(0x985B6B9064212091B4b325F68746B77262801BcB);
  priceFeed = AggregatorV3Interface(
      0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419
  );
```

Impacts

Hardcoding address variables in the contract make it difficult for it to be modified at a later stage in the contract as everything will need to be deployed again at a different address if there's a code upgrade.

Remediation

It is recommended to create dynamic functions to address upgrades so that it becomes easier for developers to make changes at a later stage if necessary.

The said function should have proper access controls to make sure only administrators can call that function using access control modifiers.

There should also be a zero address validation in the function to make sure the tokens are not lost.

If the address is supposed to be hardcoded, it is advisable to make it a constant if its value is not getting updated.

Retest:

Hardcoded address has been moved to the functional level with a setter function or constructor level

Bug ID#3 [Fixed]

Use of SafeMath

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
SafeMath library is found to be used in the contract. This increases gas consumption more than traditional methods and validations if done manually.
Also, Solidity **0.8.0** and above includes checked arithmetic operations by default, rendering SafeMath unnecessary.

Affected Code:

- contract JAY - Line 11

```
...
using SafeMath for uint256;
...
```

Impacts:
This increases the gas usage of the contract.

Remediation:
We do not recommend using the SafeMath library for all arithmetic operations. It is good practice to use explicit checks where it is really needed and to avoid extra checks where overflow/underflow is impossible.
The compiler above 0.8.0+ automatically checks for overflows and underflows.

Retest:
The use of Safe math has been removed.

Bug ID#4 [Fixed]

Dead Code

Vulnerability Type
Code With No Effects - SWC-135

Severity
Informative

Description
It is recommended to keep the production repository clean to prevent confusion and the introduction of vulnerabilities. The functions and parameters, contracts, and interfaces that are never used or called externally or from inside the contracts should be removed when the contract is deployed on the mainnet.

Affected Code

- contract JAY - function priceCheck() - Line - 113

```
function priceCheck() internal {
    // Gets there price of Jay after the TX
    uint256 newPrice = JAYtoETH(ETHinWEI);

    // Assert the new price is > than the previous price
    assert(prevPrice < newPrice);

    // Set previous price to new price
    prevPrice = newPrice;
}
```

Impacts

This does not impact the security aspect of the Smart contract but prevents confusion when the code is sent to other developers or auditors to understand and implement. This reduces the overall size of the contracts and also helps in saving gas.

Remediation

If the variables and constants are not supposed to be used anywhere, consider removing them from the contract.

Retest

The mentioned dead codes have been removed.

Bug ID#5 [Invalid Bug]

Price Manipulation by Forcing Ether in the Contract

Vulnerability Type
Business logic

Severity
Medium

Description
The contract implements some payable functions such as fallback, receive, and deposit. This could allow any more to call them and force send Ether into the contract thereby manipulating the buy and sell price during the execution of the buy/sell functions because they rely on "address(this).balance" for price calculation.

Affected Code

- contract JAY

```
    function JAYtoETH(uint256 value) public view returns (uint256) {
        return (value * address(this).balance).div(totalSupply());
    }

    function ETHtoJAY(uint256 value) public view returns (uint256) {
        return
value.mul(totalSupply()).div(address(this).balance.sub(value));
    }
```

Impacts
By force sending Ether into the contract, any external user would be able to manipulate the price of the token.

Remediation

It is recommended to modify the code logic so that it does not depend on the ETH balance of the contract for calculations of business-critical functions such as the token price.

Retest

The team informed us that this is working as intended and it is the exact business logic of the contract.

Bug ID#6 [Fixed]

Missing Events in important functions

Vulnerability Type
Missing Best Practices

Severity
Low

Description
Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log — a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions.

The contract was found to be missing these events on certain critical functions which would make it difficult or impossible to track these transactions off-chain.

Affected Code
The following functions were affected -

- contract JAY - function setMax()
- contract JAY - function setSellFee()
- contract JAY - function setBuyFee()

Impacts
Events are used to track the transactions off-chain and missing these events on critical functions makes it difficult to audit these logs if they're needed at a later stage.

Remediation
Consider emitting events for the functions mentioned above. It is also recommended to have the addresses indexed.

Retest:

Important functions are emitting events.

Bug ID#7 [Fixed]

Missing Multiple Zero Address Validations

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
The contracts were found to be setting new addresses without proper validations for zero addresses.
Address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burned forever.
Depending on the logic of the contract, this could prove fatal and the users or the contracts could lose their funds, or the ownership of the contract could be lost forever.

Affected Code

- contract JAY - function setFeeAddress()
- contract JayFeeSplitter - function setTEAMWallet()
- contract JayFeeSplitter - function setNFTWallet()
- contract JayFeeSplitter - function setLPWallet()
- contract JayLiquidityStaking - function setFeeAddress()

Impacts
If address type parameters do not include a zero-address check, contract functionality may become unavailable or tokens may be burned permanently.

Remediation
Add a zero address validation to all the functions where addresses are being set.

Retest
Zero address validation has been added.

Bug ID#8 [Fixed]

Functions should be declared External

Vulnerability Type

Gas Optimization

Severity

Gas

Description

Public functions that are never called by a contract should be declared external in order to conserve gas.

The following functions were declared as public but were not called anywhere in the contract, making public visibility useless.

Affected Code

The following functions were affected -

- contract JAY - function getBuyJay()
- contract JAY - function getSellJay()

Impacts

Smart Contracts are required to have effective Gas usage as they cost real money and each function should be monitored for the amount of gas it costs to make it gas efficient.

"**public**" functions cost more Gas than "**external**" functions.

Remediation

Use the "**external**" state visibility for functions that are never called from inside the contract.

Retest

The affected public function has been marked as external.

Bug ID#9 [Fixed]

Unnecessary Multiple Payable Functions

Vulnerability Type

Missing Best Practices

Severity

Low

Description

The contracts define multiple empty payable functions that do not serve any other purpose but to receive Ether into the contract. This is redundant and unnecessary and it is recommended to keep only one of these functions.

Affected Code

The following functions were affected -

- contract JAY - function deposit() public payable {}
- contract JAY - receive() external payable {}
- contract JAY - fallback() external payable {}
- contract JayFeeSplitter - function deposit() public payable {}
- contract JayFeeSplitter - receive() external payable {}
- contract JayFeeSplitter - fallback() external payable {}
- contract JayMart - function deposit() public payable {}
- contract JayMart - receive() external payable {}
- contract JayMart - fallback() external payable {}

Impacts

Smart Contracts are required to have effective Gas usage as they cost real money and each function should be monitored for the amount of gas it costs to make it gas efficient.
Having redundant code in the production codebase just increases the overall size and deployment costs.

Remediation

If the purpose of the contracts is just to receive Ether, it is recommended to keep only the receive() function. The deposit() function can also be kept if the developers want users to deposit Ether by calling another function.

Retest

Unnecessary functions have been removed

Bug ID#10 [Fixed]

Missing Price Feed Validation

Vulnerability Type
Input Validation

Severity
Medium

Description
Chainlink has a library **AggregatorV3Interface** with a function called **latestRoundData()**. This function returns the price feed among other details for the latest round.
The contract was found to be using **latestRoundData()** without proper input validations on the returned parameters which might result in a stale and outdated price.

Vulnerable Code

- contract JayMart - function updateFees(), function getLatestPrice() - priceFeed.latestRoundData();

```
function getLatestPrice() public view returns (int256) {
    (, int256 price, , , ) = priceFeed.latestRoundData();
    return price;
}

function updateFees()
    external
    nonReentrant
    returns (
        uint256,
        uint256,
        uint256,
        uint256
```

```
        )
    {
        // Get latest price feed
        (, int256 price, , uint256 timeStamp, ) =
priceFeed.latestRoundData();
```

Impacts

Having oracles with functions to fetch price feed without any validation might introduce erroneous or invalid price values that could result in an invalid price calculation further in the contract.

Remediation

It is recommended to have input validations for all the parameters obtained from the Chainlink price feed. Here's a sample implementation:

```
(uint80 roundID ,answer,, uint256 timestamp, uint80 answeredInRound) =
AggregatorV3Interface(chainLinkAggregatorMap[underlying]).latestRoundData();

require(answer > 0, "Chainlink price <= 0");
require(answeredInRound >= roundID, "Stale price");
require(timestamp != 0, "Round not complete");
```

Retest

Price feed validation has been added by using the new updateFees() function which has all price feed validations in place.

Bug ID#11 [Fixed]

Misspelled Variable/Typo

Vulnerability Type

Missing best practices

Severity

Informational

Description

The contract was found to be using the wrong spelling for a certain variable. These may cause confusion for future auditors or developers when reading and understanding the code.

Affected Code

- contract JayLiquidityStaking - function claim() - contactBalance

```
function claim() private returns (uint256) {
    // Retrieve pending rewards
    FEE_ADDRESS.splitFees();

    // Get the balance of the contract
    uint256 contactBalance = address(this).balance;
```

Impacts

Typo in the code may cause issues when a developer or auditor is trying to understand the code.

Remediation

It is recommended to correct the spelling of the parameters and words highlighted above.

Retest:

The spelling has been updated.

Bug ID#12 [Fixed]

Missing Input Validation in buyJay and buyNFTs

Vulnerability Type

Missing Input Validation

Severity

Low

Description

The contract accepts arrays of NFT addresses, IDs, and amounts when buying Jay and NFT. These functions don't have any input validation on the length of the array items allowing users to enter any number of inputs that may cause gas-related issues and failed transactions.

Affected Code

- contract JayMart - function buyJay(), function buyNFTs()

```
function buyJay(
    address[] calldata erc721TokenAddress,
    uint256[] calldata erc721Ids,
    address[] calldata erc1155TokenAddress,
    uint256[] calldata erc1155Ids,
    uint256[] calldata erc1155Amounts
) external payable nonReentrant {...}

function buyNFTs(
    address[] calldata erc721TokenAddress,
    uint256[] calldata erc721Ids,
    address[] calldata erc1155TokenAddress,
    uint256[] calldata erc1155Ids,
    uint256[] calldata erc1155Amounts
```

```
    ) external payable nonReentrant {...}
```

Impacts

Remediation

It is recommended to implement a maximum limit on the number of array items a user can input during their transaction.

Retest:

A max limit of 500 has been added to the array.

Bug ID#13 [Fixed]

Incorrect Documentation

Vulnerability Type
Insufficient Technical Documentation (CWE-1059)

Severity
Informational

Description
The Jay smart contracts did not have a proper documentation and the docs in the whitepaper were incorrect. The lack of proper documentation for the Jay smart contracts can make it difficult for developers and auditors to understand and interact with them.

Impacts
Lack of proper documentation for the Jay smart contracts can have a significant impact on the usability, security, and overall success of the project. Without accurate and up-to-date documentation, developers may struggle to understand the contracts and how to interact with them, which can lead to mistakes and errors in implementation.

Remediation
To address this issue, the project team should prioritize the development of comprehensive and accurate documentation for smart contracts. Additionally, the project team should communicate transparently with the community and stakeholders about the documentation issue and the steps being taken to remediate it.

Retest:
The team has partially updated the documentation. The complete work will be done soon.

6. Disclosure

The Reports provided by CredShields is not an endorsement or condemnation of any specific project or team and do not guarantee the security of any specific project. The contents of this report are not intended to be used to make decisions about buying or selling tokens, products, services, or any other assets and should not be interpreted as such.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. CredShields does not provide any warranty or representation about the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business. The report is not intended to be used as investment advice and should not be relied upon as such.

CredShields Audit team is not responsible for any decisions or actions taken by any third party based on the report.