CredShields

Smart Contract Audit

Jul 19th, 2023 • CONFIDENTIAL

Description

This document details the process and result of the JayDeriv Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of JayPeggers between June 28th, 2023, and July 5th, 2023. And a retest was performed on July 18th, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

JayPeggers

Table of Contents

CRED SHIELDS

1. Executive Summary

JayPeggers engaged CredShields to perform a smart contract audit from June 28th, 2023, to July 5th, 2023. During this timeframe, twelve (12) vulnerabilities were identified. **A retest was performed on July 18th, 2023, and all the bugs have been addressed.**

During the audit, 0 (zero) 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
JayDeriv Smart Contract	0	0	0	3	2	7	**12**
	0	**0**	**0**	**3**	**2**	**7**	**12**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in JayDeriv Smart Contract's scope during the testing window while abiding by the policies set forth by JayDeriv Smart Contract'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 June 28th, 2023, to July 5th, 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/blob/master/contracts/JayDeriv.sol - https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivFeeSplitter.sol - https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol

Table: List of Files in Scope

2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

2.1.3 Audit Goals

CredShields uses both in-house tools 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 follows 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, 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, twelve (12) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Missing NatSpec Comments	Informational	Missing best practices
Missing Zero Address Validations	Low	Missing Input Validation
Variables should be Immutable	Gas	Gas Optimization
Unnecessary Multiple Payable Functions	Low	Missing Best Practices
Superfluous Event Field	Gas	Gas Optimization
Missing 0 Value Validation	Low	Missing 0 Value Validation
Require with Empty Message	Informational	Code optimization

Gas Optimization in For Loops	Gas	Gas Optimization
Functions should be declared External	Gas	Gas Optimization
Array Length Caching	Gas	Gas Optimization
Public Constants can be Private	Gas	Gas Optimization
Unnecessary Default Value Initialization	Gas	Gas Optimization

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 July 18th, 2023, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Missing NatSpec Comments	Informational	**NA**
Missing Zero Address Validations	Low	**Fixed [18/07/2023]**
Variables should be Immutable	Gas	**Fixed [18/07/2023]**
Unnecessary Multiple Payable Functions	Low	**Fixed [18/07/2023]**
Superfluous Event Field	Gas	**Fixed [18/07/2023]**
Missing 0 Value Validation	Low	**Fixed [18/07/2023]**
Require with Empty Message	Informational	**Fixed [18/07/2023]**
Gas Optimization in For Loops	Gas	**Fixed [18/07/2023]**

Functions should be declared External	Gas	**Fixed [18/07/2023]**
Array Length Caching	Gas	**Fixed [18/07/2023]**
Public Constants can be Private	Gas	**Fixed [18/07/2023]**
Unnecessary Default Value Initialization	Gas	**Fixed [18/07/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [NA]

Missing NatSpec Comments

Vulnerability Type
Missing best practices

Severity
Informational

Description:
Solidity contracts use a special form of comments to document code. This special form is named the Ethereum Natural Language Specification Format (NatSpec).
The document is divided into descriptions for developers and end-users along with the title and the author.
The contracts in the scope were missing these comments.

Impacts:
Without Natspec comments, it can be challenging for other developers to understand the code's intended behavior and purpose. This can lead to errors or bugs in the code, making it difficult to maintain and update the codebase. Additionally, it can make it harder for auditors to evaluate the code for security vulnerabilities, increasing the risk of potential exploits.

Remediation:
Developers should review their codebase and add Natspec comments to all relevant functions, variables, and events. Natspec comments should include a description of the function or event, its parameters, and its return values.

Retest

The team decided not to add comments as of now which is fine as it is not exploitable and just an informational issue to make the code more readable.

Bug ID #2 [Fixed]

Missing 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 Variables and Line Numbers

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivFeeSplitter.sol#L23 - rewardToken
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol#L47-L49 _liquidityToken, _rewardToken, _backingToken

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
0 address validation has been added.

- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDerivFeeSplitter.sol#L24
- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDerivLiquidityStaking.sol#L47-L49

Bug ID #3 [Fixed]

Variables should be Immutable

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
Declaring state variables that are not updated following deployment as immutable can save gas costs in smart contract deployments and function executions. Immutable state variables are those that cannot be changed once they are initialized, and their values are set permanently.

By declaring state variables as immutable, the compiler can optimize their storage in a way that reduces gas costs. Specifically, the compiler can store the value directly in the bytecode of the contract, rather than in storage, which is a more expensive operation.

Affected Code:

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivFeeSplitter.sol#L20 - MIN

Impacts:
Gas usage is increased if the variables that are not updated outside of the constructor are not set as immutable.

Remediation:
An "`immutable`" attribute should be added in the parameters that are never updated outside of the constructor to save the gas.

Retest

MIN has been set as immutable - https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDerivFeeSplitter.sol#L20

Bug ID #4 [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 -

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L175-L179

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
The functions have been removed.

Bug ID #5 [Fixed]

Superfluous Event Field

Vulnerability Type
Gas Optimization

Severity
Gas

Description
“block.timestamp” and “block.number” are by default added to event information. Adding them manually costs extra gas.

Affected Code

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L83
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L100
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L114
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L131

Impacts
Emitting block values in events costs extra gas as it is not required and is available by default.

Remediation
“block.timestamp” do not need to be added manually. Consider removing it from the emitted events.

Retest
The events are not emitting block values such as block.timestamp.

Bug ID #6 [Fixed]

Missing 0 Value Validation

Vulnerability Type
Input Validation

Severity
Low

Description
The contract has functions to sell NFT by supplying the number of Jay tokens. This function lacks input validation on the amount of tokens which can then be set as 0 leading to unexpected result.

Affected Code

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L69
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L102

Impacts
Missing 0 value validation could allow unintentional results such as burning 0 tokens, and transferring 0 amount that could pollute the events.

Remediation
It is recommended to have a threshold validation for the token amount such as keeping it above 0.

Retest
Boundary value checks have been added.

- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDeriv.sol#L68

- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDeriv.sol#L103

Bug ID #7 [Fixed]

Require with Empty Message

Vulnerability Type
Code optimization

Severity
Informational

Description
During analysis; multiple **require** statements were detected with empty messages. The statement takes two parameters, and the message part is optional. This is shown to the user when and if the **require** statement evaluates to false. This message gives more information about the conditional and why it gave a false response.

Affected Code

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivFeeSplitter.sol#L44
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivFeeSplitter.sol#L49
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivFeeSplitter.sol#L53
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivFeeSplitter.sol#L58
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L52
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L86
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L119
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L143

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L148-L149
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L155
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol#L54
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol#L159

Impacts

Having a short descriptive message in the **require** statement gives users and developers more details as to why the conditional statement failed and helps in debugging the transactions.

Remediation

It is recommended to add a descriptive message, no longer than 32 bytes, inside the **require** statement to give more detail to the user about why the condition failed.

Retest

The require statements are reverting with descriptive text.

Bug ID #8 [Fixed]

Gas Optimization in For Loops

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Loops are in most cases bounded by definition (the bounding is represented by the exit condition). Therefore in the vast majority of cases, checking for overflows is really not needed, and can get very gas expensive. Here's an example:

```
pragma solidity ^0.8.0;

contract Test1 {


    function loop() public pure {

        for(uint256 i = 0; i < 100; i++) {
        }

    }
}
```

```
pragma solidity ^0.8.0;

contract Test {

    function loop() public pure {

        for(uint256 i = 0; i < 100;) {

            unchecked {

                i++;

            }

        }

    }

}
```

loop() in Test1 costs more than 31K gas, vs 25.5K gas for loop() in Test2.

Affected Code

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol#L66

Impacts

Removing overflow validations using unchecked blocks will save gas in the loops.

Remediation

It is recommended to implement unchecked blocks in for loops wherever possible since they are already bounded by an upper length and there's a very rare chance that it might overflow.

Retest

The loop has been optimized - https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDerivLiquidityStaking.sol#L70-L81

Bug ID #9 [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

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L63-L66 - setMax
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L51-L56 - init
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L47-L49 - setJAYNFT
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol#L81-L83 - setStart
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol#L58-L77 - initalize

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 public functions have been updated as external.

- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDeriv.sol#L45-L54
- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDeriv.sol#L61-L64
- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDerivLiquidityStaking.sol#L61-L91

Bug ID #10 [Fixed]

Array Length Caching

Vulnerability Type
Gas Optimization

Severity
Gas

Description
During each iteration of the loop, reading the length of the array uses more gas than is necessary. In the most favorable scenario, in which the length is read from a memory variable, storing the array length in the stack can save about 3 gas per iteration. In the least favorable scenario, in which external calls are made during each iteration, the amount of gas wasted can be significant.

Affected Code

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol#L66

Impacts
Reading the length of an array multiple times in a loop by calling .length costs more gas.

Remediation
Consider storing the array length of the variable before the loop and use the stored length instead of fetching it in each iteration.

Retest
Array length caching has been implemented to save gas - https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDerivLiquidityStaking.sol#L69

Bug ID #11 [Fixed]

Public Constants can be Private

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Public constant variables cost more gas because the EVM automatically creates getter functions for them and adds entries to the method ID table. The values can be read from the source code instead.

Affected Code

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L21
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L27
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L29
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L33

Impacts
Public constants are more costly due to the default getter functions created for them, increasing the overall gas cost.

Remediation
If reading the values for the constants is not necessary, consider changing the public visibility to private.

Retest

The variables have been made private to save gas.

- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDeriv.sol#L21
- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDeriv.sol#L27
- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDeriv.sol#L29
- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDeriv.sol#L33

Bug ID #12 [Fixed]

Unnecessary Default Value Initialization

Vulnerability Type
Gas Optimization

Severity
Gas

Description
In Solidity, data types are automatically assigned default values if they are not explicitly initialized. However, the initialization of default values can sometimes be unnecessary or inefficient, depending on the specific use case.

Affected Code

- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol#L38
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDerivLiquidityStaking.sol#L40
- https://github.com/toshimon-io/jay-contracts/blob/master/contracts/JayDeriv.sol#L31

Impacts
Initializing data types to their default values is not required and costs additional gas.

Remediation
It's not recommended to initialize the data types to their default values unless there's a use-case because it's unnecessary and costs around ~3 gas.

Retest
This is fixed. The default value is not reinitialized anymore.

- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDeriv.sol#L31

- https://github.com/toshimon-io/jay-contracts/blob/f45e205cdaa9c4aa3866266c762923500b280a4b/contracts/JayDerivLiquidityStaking.sol#L38-L40

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.