#Hashlock.

Jubi DAO (DAO)

SMART CONTRACT

Security Audit

Performed on Contracts:

Venture.sol
Types.sol
NftAllocator.sol
Factory.sol
ERC20FixedPriceAllocator.sol
ERC20FixedPriceAllocatorSignatureStore.sol
SignatureStore
Commit Hash: 5b44d40068814335a29c093fab2f39a593f35d19
Commit Hash: 6e394573c531c4386a6f2c7a1212a94da2d320f2

Platform

ETH

hashlock.com.au

NOVEMBER 2023

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

The Jubi Dao team partnered with Hashlock to conduct a security audit of their protocol smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that it is ready for mainnet deployment.

Project Context

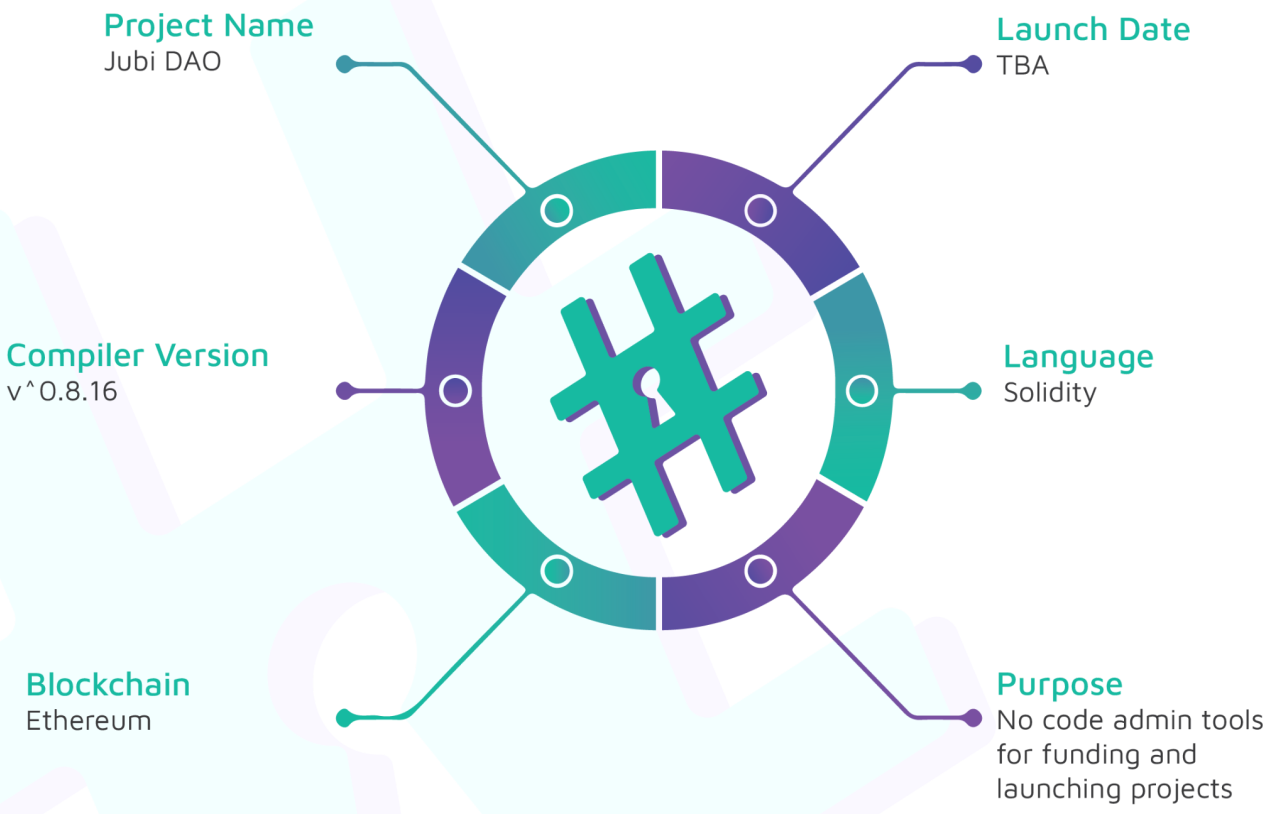
Jubi Dao is a soon-to-launch protocol aiming to be an end-to-end manager of all admin and launch processes within a protocol. The Jubi Dao team engaged Hashlock to ensure that their protocol is ready for launch.

Project Name: Jubi Dao
Contract Address: N/A
Compiler Version: ^0.8.16

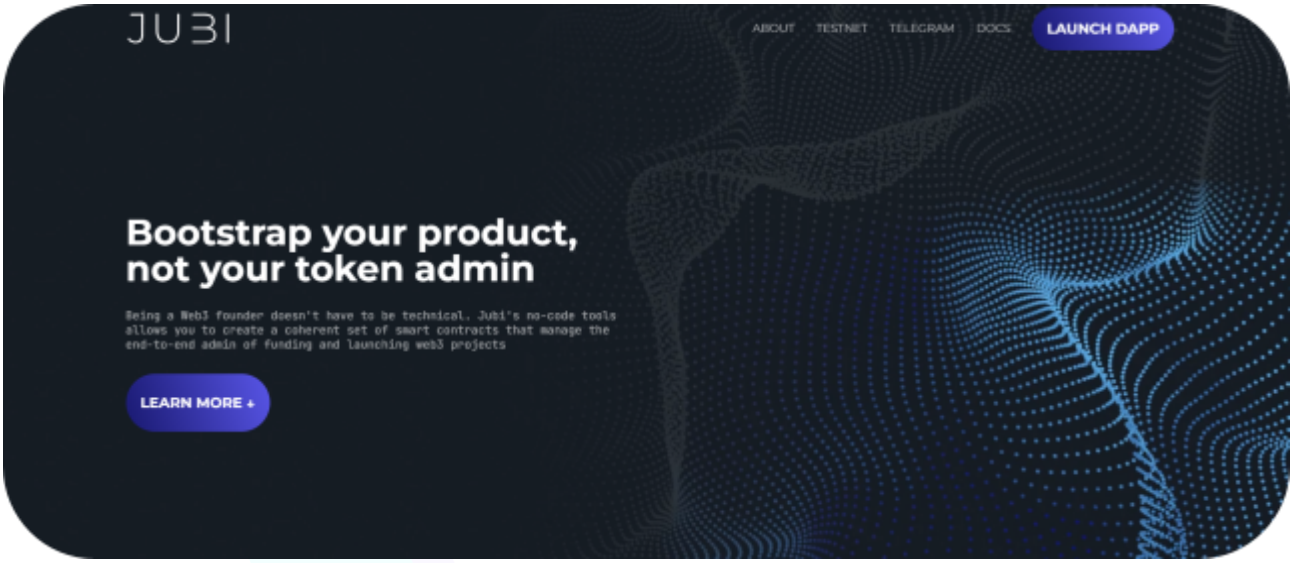
Logo:

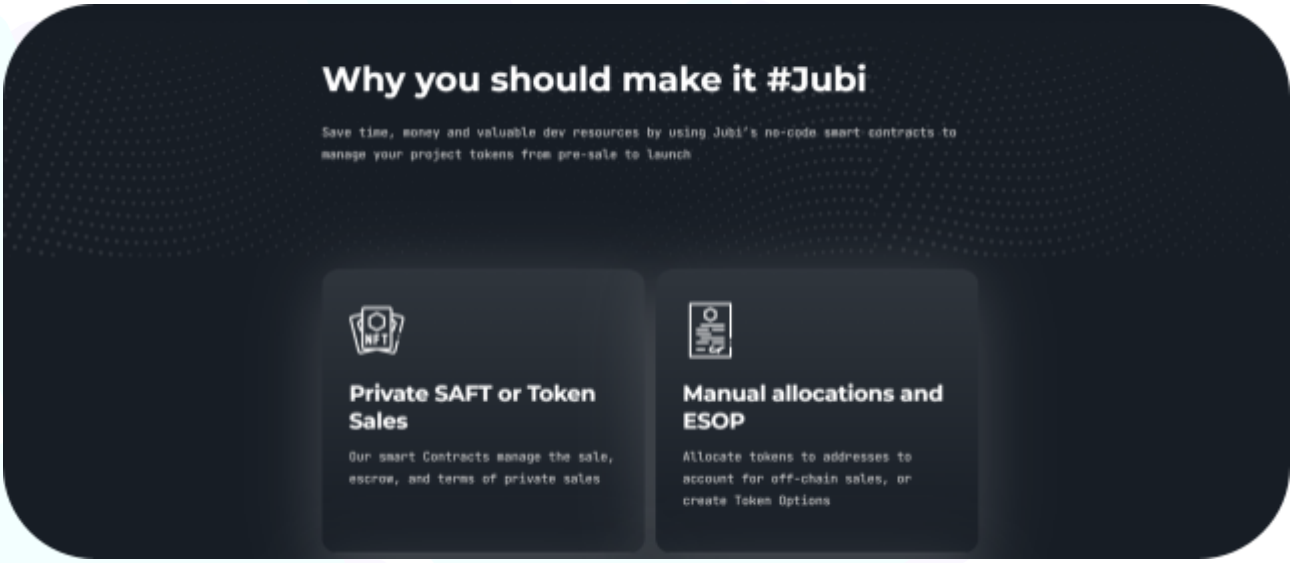
JUBI

Visualised Context:



Project Visuals:





Jubi Dao Project Leadership

Title	Name	Photo	Links
Founder	Ash van der Spuy		https://www.linkedin.com/in/ashvds/
Founder	Harrison Dell		https://www.linkedin.com/in/harrison-dell-0b220111b/

Audit scope

We at Hashlock audited the solidity code within the Jubi Dao Project, the scope of works included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line by line analysis and were supported by software assisted testing.

Description	**Project Review and Security Analysis Report for Jubi DAO Protocol Smart Contracts and other factors.**
Platform	**Ethereum / Solidity**
Audit Date	**June, 2023**
Contract 1	ERC20FixedPriceAllocator.sol
Contract 1 MD5 Hash	BB6BBB03A1B7FC20AD11C01474EC1D0C
Contract 2	Factory.sol
Contract 2 MD5 Hash	7AFA0C63E590CD143DA38B735D5A9FDF
Contract 3	NftAllocator.sol
Contract 3 MD5 Hash	528687A210782E499D8576F960C5D99
Contract 4	Venture.sol
Contract 4 MD5 Hash	CE1C29EC5152921D9D296F9CA470C1A2

Description	Project Review and Security Analysis Report for Jubi DAO Protocol Additional Smart Contracts and other factors.
Platform	**Ethereum / Solidity**
Audit Date	**September, 2023**
Contract 1	ERC20OptionsAllocator.sol
Contract 1 MD5 Hash	8eeb44f54df404c60d7308b048b7c567
Contract 2	ERC20ManualAllocator.sol
Contract 2 MD5 Hash	6a974d357c37486cafb58c1f7420ea79
Contract 3	JubiERC20BasicImpl.sol
Contract 3 MD5 Hash	294295942169836916e8c29cdee2a7be
Contract 4	JubiERC20GovernanceImpl.sol
Contract 4 MD5 Hash	fd32d2c020118468ba7f6608085a6d6b
Contract 5	JubiERC20UltimateImpl.sol
Contract 5 MD5 Hash	36061f0c81f2a2bfc2daca6eb10dadff
Contract 6	BasicTokenCreator.sol
Contract 6 MD5 Hash	46ab20320394ad99567c5a0be2961cc9
Contract 7	GovernanceTokenCreator.sol
Contract 7 MD5 Hash	fd32d2c020118468ba7f6608085a6d6b
Contract 8	UltimateTokenCreator.sol
Contract 7 MD5 Hash	36061f0c81f2a2bfc2daca6eb10dadff

Security Rating

After the resolution of our findings, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts. We have identified some high severity and other vulnerabilities that will need to be addressed before mainnet launch.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in the Function list section and all identified issues can be found in the Audit overview section.

All vulnerabilities initially identified have been addressed.

Hashlock found:

2 High severity vulnerabilities

8 Medium severity vulnerabilities

13 Low severity vulnerabilities

8 Gas findings

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

Standardised Checks

Main Category	Subcategory	Result
General Code Checks	Solidity/compiler version stated	Passed
	Consistent pragma version across each contract	Passed
	Outdated Solidity Version	Reviewed
	Overflow/underflow	Passed
	Correct checks, effects, interaction order	Reviewed
	Lack of check on input parameters	Reviewed
	Function input parameters check bypass	Passed
	Correct Access control	Reviewed
	Built in emergency features	Passed
	Correct event logs	Reviewed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Reviewed
	Features claimed	Passed
	delegatecall() vulnerabilities	Passed
	Other programming issues	Reviewed
Code Specification	Correctly declared function visibility	Passed
	Correctly declared variable storage location	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Reviewed
Gas Optimization	"Out of Gas" Issue	Passed
	High consumption 'for/while' loop	Reviewed

	High consumption ‘storage’ storage	Passed
	Assert() misuse	Passed
Tokenomics Risk	The maximum limit for mintage not set	Reviewed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Preliminary Audit Result: VULNERABLE

Revised Audit Result: PASSED

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
File 1 ERC20FixedPriceAllocator.sol Other Specifications: This contract allows Ventures to sell tokens and allocate them to purchase accounts according to a distribution schedule. The Jubi fee amount: 1 token taken as fee.	**Contract achieves this functionality.**
File 2 Factory.sol Owner has control over following functions: Updates the address of an allocator. Updates the address of ventureImp. Updates the address of the Signature store. Updates the address of ERC20 Fixed Price Signature store. Revert when `msg.sender` is not authorised to upgrade the contract. Other Specifications: Factory contracts used to create new ventures and allocators.	**Contract achieves this functionality.**
File 3 NftAllocator.sol	**Contract achieves this**

Admin has control over following functions: NftAllocator admin can receive erc721 tokens. NftAllocator:admin can migrate current venture to a new venture manager `_newVenture`. Other Specifications: NftAllocator contracts are used to invest and claim a NFT.	**functionality.**
File 4 Venture.sol Owner/Admin has control over following functions: Venture admin can add an allocator: `newAllocator` of type: `_type` to this venture. Venture admin can unallocate tokens from an allocator and decrease totalAllocated for the token. The ventureToken values can be set by the admin. A new admin address can be allocated by the current owner. Admin address can be removed by the owner. Other Specifications: Venture contracts are used to manage ventures.	**Contract achieves this functionality.**
File 5 ERC20ManualAllocator.sol	**Contract achieves this functionality.**

Other Specifications: This contract allows a Venture to create an allocation of Venture tokens, and will allocate the tokens to the relevant accounts according to a distribution schedule. This is a partner to ERC20FixedPriceAllocator, but assumes that contracts and funds exchange happened off-chaib so it only manages the distribution of tokens	
File 6 ERC20OptionsAllocator.sol Other Specifications: This contract allows a Venture to Venture token options to addresses, and will allow accounts to execute options according to a vesting schedule	**Contract achieves this functionality.**
File 7 JubiERC20BasicImpl.sol Other Specifications: This contract is used as a template for ERC20 tokens in the jubi.io ecosystem.	**Contract achieves this functionality.**
File 8 JubiERC20GovernanceImpl.sol Other Specifications: This contract is used as a template for ERC20 tokens in the jubi.io ecosystem.	**Contract achieves this functionality.**
File 9 JubiERC20UltimateImpl.sol Other Specifications:	**Contract achieves this functionality.**

This contract is used as a template for ERC20 tokens in the jubi.io ecosystem.	
File 10 BasicTokenCreator.sol Other Specifications: This contract is used to create new basic tokens.	**Contract achieves this functionality.**
File 11 GovernanceTokenCreator.sol Other Specifications: This contract is used to create new governance tokens.	**Contract achieves this functionality.**
File 12 UltimateTokenCreator.sol Other Specifications: This contract is used to create new ultimate tokens. Ultimate tokens allow for governance rights as well as snapshotting and ERC-20 `permit`.	**Contract achieves this functionality.**

Function List

Jubi Dao Contracts

- From Factory
- From Venture
- From NftAllocator
- From ERC20FixedPriceAllocator

- From ERC20OptionsAllocator
- From ERC20ManualAllocator
- From JubiERC20 Tokens
- From Token Creators

Functions

	Functions	Visibility	Observation	Conclusion
1	constructor - From Factory	Public		No Issue
2	initialize - From Factory	Public	initializer	No Issue
4	initializeAtUpgrade - From Factory	External	onlyOwner	No Issue
7	createVenture - From Factory	External		No Issue
8	createERC20FixedPriceAllocator - From Factory	External		No Issue
9	getVentures - From Factory	External		No Issue
10	updateAllocatorImpl - From Factory	External	onlyOwner	No Issue
11	updateVentureImpl - From Factory	External	onlyOwner	No Issue
12	updateSignatureStoreNftAllocatorImpl - From Factory	External	onlyOwner	No Issue
13	updateERC20FixedPriceAllocatorSignatureStoreImpl - From Factory	External	onlyOwner	No Issue
14	_updateAllocatorImpl - From Factory	Internal	onlyOwner	No Issue

15	_updateERC20FixedPriceAllocatorSignatureStoreImpl - From Factory	Internal	onlyOwner	No Issue
16	_authorizeUpgrade - From Factory	Internal	onlyOwner	No Issue
17	initialize - From Venture	External	initializer	No Issue
18	addAllocator - From Venture	External	onlyAllocatorAttacher	No Issue
19	setVentureToken - From Venture	External	onlyAdmin	No Issue
20	setVentureTokenSupply - From Venture	External	onlyAdmin	No Issue
21	getAllocators - From Venture	External		No Issue
22	addAdmin - From Venture	External	onlyOwner	No Issue
23	removeAdmin - From Venture	External	onlyOwner	No Issue
24	addAllocatorManager - From Venture	External	onlyAdmin	No Issue
25	removeAllocatorManager - From Venture	External	onlyAdmin	No Issue
26	isAdmin - From Venture	External		No Issue
27	isAdminOrAllocatorManager - From Venture	External		Fixed
28	_removeAdmin - From Venture	Private	onlyAdmin	No Issue
29	_addAdmin - From Venture	Private	onlyAdmin	No Issue
30	_addAllocatorManager - From Venture	Private	onlyAdmin	No Issue
31	_addAllocatorAttacher - From Venture	Private	onlyAllocatorAttacher	No Issue
32	_removeAllocatorManager - From Venture	Private	onlyAdmin	No Issue
33	initialize - From NftAllocator	External		No Issue
34	invest - From NftAllocator	External		Fixed
35	claim - From NftAllocator	External		No Issue

36	close - From NftAllocator	External	isAdminOrAllocator Manager	No Issue
37	setNft - From NftAllocator	External	isAdminOrAllocator Manager	No Issue
38	migrateVenture - From NftAllocator	External	onlyAdmin	No Issue
39	onERC721Received - From NftAllocator	External		No Issue
40	initialize - From ERC20FixedPriceAllocator	External	initializer	No Issue
41	purchase - From ERC20FixedPriceAllocator	External		No Issue
42	claim - From ERC20FixedPriceAllocator	External		No Issue
43	claimEscrow - From ERC20FixedPriceAllocator	External	isAdminOrAllocator Manager	No Issue
44	close - From ERC20FixedPriceAllocator	External	isAdminOrAllocator Manager	No Issue
45	_close - From ERC20FixedPriceAllocator	Internal	isAdminOrAllocator Manager	No Issue
46	calculateReleased - From ERC20FixedPriceAllocator	External		No Issue
47	setAllocationToken - From ERC20FixedPriceAllocator	External	isAdminOrAllocator Manager	No Issue
48	migrateVenture- From ERC20FixedPriceAllocator	External	onlyAdmin	No Issue
49	calculateTotalTokensReleased - From ERC20FixedPriceAllocator	External		No Issue
50	allocationToPurchaseToken - From ERC20FixedPriceAllocator	Internal		Fixed
51	initialize - From ERC20OptionsAllocator	External	initializer	No Issue
52	allocate - From ERC20OptionsAllocator	External		No Issue
53	allocateBulk - From ERC20OptionsAllocator	External		Fixed

54	_allocate - From ERC20OptionsAllocator	Private		No Issue
55	terminateVesting - From ERC20OptionsAllocator	External		Fixed
56	executeOption - From ERC20OptionsAllocator	External	nonReentrant	No Issue
57	close - From ERC20OptionsAllocator	External		No Issue
58	_close - From ERC20OptionsAllocator	Private		No Issue
59	calculateVested - From ERC20OptionsAllocator	Public		No Issue
60	calculateVestedAt - From ERC20OptionsAllocator	Public		No Issue
61	calculateWillNotVest - From ERC20OptionsAllocator	Public		No Issue
62	setAllocationToken - From ERC20OptionsAllocator	External		Fixed
63	migrateVenture - From ERC20OptionsAllocator	External		No Issue
64	allocationToPurchaseToken - From ERC20OptionsAllocator	Internal		Fixed
65	initialize - From ERC20ManualAllocator	External	initializer	No Issue
66	allocate - From ERC20ManualAllocator	External		No Issue
67	allocateBulk - From ERC20ManualAllocator	External		No Issue
68	_allocate - From ERC20ManualAllocator	Private		No Issue
69	claim - From ERC20ManualAllocator	External	nonReentrant	No Issue
70	close - From ERC20ManualAllocator	External		No Issue
71	_close - From ERC20ManualAllocator	Internal		No Issue

72	calculateReleased - From ERC20ManualAllocator	Public		No Issue
73	setAllocationToken - From ERC20ManualAllocator	External		No Issue
74	migrateVenture - From ERC20ManualAllocator	External		No Issue
75	constructor - From JubiERC20 Tokens	Public		No Issue
76	pause - From JubiERC20 Tokens	Public	onlyOwner	No Issue
77	unpause - From JubiERC20 Tokens	Public	onlyOwner	No Issue
78	setMinter - From JubiERC20 Tokens	External	onlyOwner	No Issue
79	mint - From JubiERC20 Tokens	Public	onlyMinter	No Issue
80	burnFrom - From JubiERC20 Tokens	Public	onlyMinter	No Issue
81	burn - From JubiERC20 Tokens	Public	onlyMinter	No Issue
82	_beforeTokenTransfer - From JubiERC20 Tokens	Internal		No Issue
83	recoverERC20 - From JubiERC20 Tokens	Public	onlyOwner	No Issue
84	_afterTokenTransfer - From JubiERC20 Tokens	Internal		No Issue
85	_mint - From JubiERC20 Tokens	Internal		No Issue
86	_burn - From JubiERC20 Tokens	Internal		No Issue
87	_requireMinter - From JubiERC20 Tokens	Internal		No Issue
88	Initialize - From Token Creators	Public	initializer	No Issue
89	createToken - From Token Creators	Public		No Issue
90	_authorizeUpgrade - From Token Creators	Internal	onlyOwner	No Issue

Code Quality

This audit scope involves the solidity smart contracts of the Jubi Dao project, as outlined in the Audit Scope section. All contracts, libraries and interfaces intend to follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however some refactoring is required.

The code is extremely well commented and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Jubi Dao Protocol's smart contract code in the form of Github access.

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

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well known industry standard open source projects.

Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

Significance	Description
High	High severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues and inefficiencies

Audit Findings

High

[H-01] NftAllocator#invest - Function is prone to reentrancy attacks

Description

The `invest` function within the `NftAllocator` contract has a vulnerability to reentrancy attacks due to not following the recommended Checks Effects Interactions pattern.

Vulnerability Details

The current implementation allows an authorised investor to exceed their designated maximum investment limit by employing a smart contract, primarily due to the placement of the updates to the `invitesAllocation[inviteCode].claimed` and `investmentCount` variables. These variables should be updated before the `safeTransferFrom` call is executed.

Proof of Concept

1. Consider a scenario where an investor (possessing an invite code for a maximum of 5 tokens) calls the invest function to claim 5 tokens.
2. After the first NFT is transferred to the user, the investor, by leveraging onERC721Received, calls the invest function again, with 5 tokens. At this point, neither the invitesAllocation[inviteCode].claimed nor the investmentsCount variables have been updated, since their updates occur after the call.
3. The invest function for five additional tokens to the investor is successful. As the variables have yet to be updated, the system is able to bypass the checks.
4. The remaining four tokens from the initial call are then claimed, leading the investor to accumulate a total of 10 tokens, which exceeds their maximum limit of 5. The variables are now updated to 5 tokens claimed however these extra tokens still get approved, as they were part of the initial call that passed the initial requisite checks and the operation continued from where it had left off (in the for loop).

Impact

An investor can claim more tokens than intended.

Recommendation

To rectify this vulnerability, add a reentrant modifier as well as update all relevant variables prior to making the external call.

Status

Resolved

[H-02] ERC20OptionsAllocator#allocationToPurchaseToken - Purchase price is lower than intended due to precision loss

> **Auditor's Note:** This finding also applies to `ERC20FixedPriceAllocator`

Description

The `allocationToPurchaseToken` function does division before multiplication. This leads to precision loss which results in a lower (or even zero) purchase price.

Vulnerability Details

The `allocationToPurchaseToken` function is used to calculate the amount of `purchaseToken` that the user has to send to the `venture` contract when exercising their options.

```
    function allocationToPurchaseToken(
        uint256 _allocation
    ) internal view returns (uint256) {
        return (_allocation / 1e18) * strikePrice;
    }
```

As shown above, the `strikePrice` is multiplied with the result of `(_allocation / 1e18)`. This will result in great precision loss, especially when `_allocation` is small and `strikePrice` is large.

Furthermore, in cases where `_allocation < 1e18`, the result of `(_allocation / 1e18) * strikePrice` will be `0`, which means that the user is able to purchase the tokens for free.

Impact

The cost of exercising the options is a lot lower than intended, and the user can receive tokens for free.

Recommendation

Perform multiplication before division.

Status

Resolved

Medium

[M-01] Unsafe downcasting in NftAllocator.invest() could result in integer truncation

Description

The `NftAllocator` contract allows users to invest a certain amount of NFT tokens and claim venture tokens in return. When the user calls invest, a `uint256 numTokens` are passed in which is downcasted to a `uint128` when appending to `invitesAllocation[inviteCode].claim` however, this can be deemed unsafe should the `numTokens` value get too large.

Vulnerability Details

Solidity downcasting is referred to as the process of converting a variable of a certain type to a smaller size or range. Downcasting a larger uint to a smaller one is deemed unsafe if certain edge case scenarios are met for instance, if the variable being downcasted exceeds maximum bits of the target type. This may occur if the user wants to claim a very large amount of allocated invites.

Impact

If `numTokens` is too large, the value of `invitesAllocation[inviteCode].claim` can be incorrect.

Recommendation

It's recommended that uint256's are left consistent when performing mathematical operations. If this is a core requisite to the functionality of the protocol, It's recommended that OpenZeppelin's `SafeCast` is used to downcast variables to their target type.

Status

Resolved

[M-02] Usage of tx.origin for authentication could allow for phishing attacks when adding an allocator

Description

The `Venture` contract allows administrators to manage their newly deployed blockchain infrastructure through the adding of allocators and management of their administrative roles. The `addAllocator` and `markUnallocatedTokensReturned` uses the `_isAdminOrAllocationManager` is called and `tx.origin` is used to authenticate the admin or allocation manager however, the use of `tx.origin` in authentication may enable phishing attacks against the contract.

Vulnerability Details

The key difference between `msg.sender` and `tx.origin` is that the `msg.sender` is the direct user or contract to interact with the `Venture` contract while `tx.origin` is the EOA who triggered the transaction. A malicious user could insert a proxy between the admin

or allocation manager to add an allocator or mark unallocated tokens as returned without authorization.

The users will be interacting with the factory contract to add an allocator, which means `msg.sender` is the factory. It should be the user that initiated, hence `tx.origin`.

The protocol team just needs to consider both scenarios if the users want to interact with the contract directly or with the factory, or even locking off the contract to only be accessible to the factory altogether.

Impact

A malicious user could insert a proxy between the admin or allocation manager to add an allocator or mark unallocated tokens as returned without authorization.

Recommendation

It's recommended that if the factory is the caller, the factory could pass the `msg.sender` received as a parameter which can be used to authenticate. If the factory is not the caller, authenticate `msg.sender` directly and ensure the newly introduced account `parameter == msg.sender`.

Status

Resolved

[M-03] The venture admin can influence fees sent to Jubi by setting the absolute minimum fee

Description

The venture deployed by venture admins (both ERC-20 and ERC-721) allows certain parameters to be set which includes the Jubi fee (a fee being transferred to JubiDAO for usage of their infrastructure). This fee is being controlled by the venture admin and therefore, Jubi may not receive the correct fees.

Vulnerability Details

Let us consider the line of code in the ERC20FixedPriceAllocator when initializing the contract:

```
jubiFee = (1e18 * _jubiFeePercent.num) / _jubiFeePercent.den;
```

The Jubi numerator and Jubi denominator are both user supplied which may allow an admin to set the fee to `1e18 * 1e18 / 1e18`. Depending on the context of the venture this fee may be miniscule resulting in missed fees for Jubi.

Impact

Jubi may not receive the correct amount of fees.

Recommendation

It's recommended that Jubi set venture fees in the factory when deploying contracts, and allowing Jubi admins to set their fees taken from each protocol using their

infrastructure. However, fees should be limited to no more than 25% of the total token value of each purchase to prevent protocol admins from setting fees to > 100%.

Status

Resolved

[M-04] - Reentrancy vulnerability in the claim function of the NftAllocator resulting in theft of ERC-1155 NFT tokens

Description

The `NftAllocator` allows venture capitalists to set up an allocator and initiate an initial token offering to people with supplied invite codes. Users can invest their tokens via the `invest()` function and claim their NFTs via the `claim()` function if the amount of NFTs to claim is not enough. Because the checks-effects interactions pattern is not implemented in the claim function, this could open the protocol up to a reentrancy vulnerability.

Vulnerability Details

While normal NFT's might not be able to exploit this vulnerability due to a revert as a result of token already claimed, ERC-1155 tokens may be able to be stolen should the venturer choose to use such tokens. ERC-1155 tokens are a combination of ERC-20 and ERC-721 tokens where there could be multiple of one token. A good use case for this would be gaming assets. Let us consider a protocol owner of a gaming company. They wish to start their venture using Jubi's infrastructure to sell in-game items, sword and shield for simplicity. Let sword be NFT ID 0 shield be NFT ID 1 where the venturer wishes to sell 100 sword tokens and 100 shield tokens. A malicious attacker may exploit the reentrancy issue to steal multiple sword or shield tokens because the next id has not been incremented yet. This vulnerability is only applicable if future ventures decide to use ERC-1155 tokens.

Recommendations

It's recommended that the non reentrant modifier is used from OpenZeppelin's contract libraries to lock up functions during execution.

Status

Acknowledged

> JubiDAO has mentioned that the contracts will not use ERC-1155 tokens.

[M-05] ERC20OptionsAllocator#setAllocationToken - Impossible to terminate vesting if allocation token is set late

Description

If the allocator contract sets an `allocationToken` after the original `vestingScheduleStartTimeStamp` value, it may not be possible to call `terminateVesting` as the `vestingEndDate` is not updated to reflect the new `vestingScheduleStartTimeStamp` value.

Vulnerability Details

The `setAllocationToken` function is used to set the `allocationToken` after the contract is deployed. In cases where the `allocationToken` minting did not happen on schedule, the venture admin has to restart the vesting schedule from the current timestamp. When this happens, `vestingEndDate` is not updated to reflect the new `vestingScheduleStartTimeStamp` value when this is done.

```
    /**
     * @notice Sets the token address for the token that was sold and transfers the
amount required to fulfill all claims to this contract
     * @param _allocationToken The token that was sold using this contract, the
required amount will be transferred to the contract
     * @param _setReleaseScheduleStart If set to true, starts the distribution
schedule
     * @dev Use '_setClaimableScheduleStart' if token minting did not happen on
schedule, to start the schedule since the token is now available
     */
    function setAllocationToken(
        IERC20 _allocationToken,
        bool _setReleaseScheduleStart
    ) external {
        ...
        if (_setReleaseScheduleStart) {
            vestingScheduleStartTimeStamp = block.timestamp;
        }
        allocationToken = _allocationToken;
    }
```

The `terminateVesting` function performs a check to make sure that vesting hasn't ended.

```
    function terminateVesting(address account) external {
        ...
        require(
            vestingEndDate > block.timestamp,
            "Allocator: Vesting period ended"
        );
        ...
    }
```

Since the `vestingEndDate` value was not updated, this check may fail in the middle of vesting when it should pass.

Impact

It may be impossible to terminate the vesting of an account, depending on the time that the `terminateVesting` function is called.

Recommendation

Update `vestingEndDate` along with `vestingScheduleStartTimeStamp` in `setAllocationToken`.

```
    function setAllocationToken(
        IERC20 _allocationToken,
```

```
        bool _setReleaseScheduleStart
    ) external {
        ...
        if (_setReleaseScheduleStart) {
            vestingScheduleStartTimeStamp = block.timestamp;
+           vestingEndDate = block.timestamp + vestingDuration;
        }
        allocationToken = _allocationToken;
    }
```

Status

Resolved

[M-06] ERC20OptionsAllocator#terminateVesting - Terminating an account's vesting more than once can result in insolvency

Description

There is no restriction on how many times a user's vesting can be terminated. When terminating more than once per user, more tokens will be transferred to `venture.fundsAddress()` than intended, resulting in insolvency of the allocator contract.

Vulnerability Details

The `terminateVesting` function calculates the amount of tokens that will not be vested through `calculateWillNotVest(account)` and returns those `allocationTokens` back to the venture funds address.

```
    function calculateWillNotVest(
        address account
    ) public view returns (uint256 unvestedAmount) {
        ...
        return allocation[account] - calculateVestedAt(account,
vestingStopped[account]);
    }
```

If `terminateVesting` is called again for the same user at a future timestamp, more tokens will be sent back to the venture funds address, even when all of the user's never-vested tokens have already been sent.

Proof of Concept

Consider the scenario where there are two allocated accounts `accA` and `accB` both with 10 allocated tokens.

At time of termination `t1`, `accA`'s number of vested tokens is 1. This means the refund when calling `terminateVesting` is 9 tokens.

However, if the same account was terminated vesting again at `t2`, the number of vested tokens is now 2, which equals to another refund of 8 tokens. In total, this is a refund of 17 tokens, which is above `accA`'s 10 allocated tokens.

The 7 outstanding tokens come out of `accB`'s allocation, as they're now left with only 3 tokens to claim. Depending on when `accB` decides to claim, their `calculateVested`

amount may be greater than 3 tokens, which would result in them being unable to claim any tokens as there's insufficient `allocationToken` balance for the `safeTransfer` to occur.

Impact

More tokens get sent back to the venture funds address than intended. The allocator contract may not have enough funds to fulfil exercised options, resulting in insolvency.

Recommendation

Restrict `terminateVesting` so that an account can only be terminated once.

Status

Resolved

[M-07] Venture#markUnallocatedTokensReturned - Function can only be called once per allocator

Description

The `Venture.markUnallocatedTokensReturned` function can only be called once per allocator due to an incorrect check. This affects `ERC20OptionsAllocator`, which may need to perform multiple external calls to that function.

Vulnerability Details

The `venture.markUnallocatedTokensReturned` external call happens inside the `close` and `terminateVesting` functions inside `ERC20OptionsAllocator`. However, for every allocator address, this external call can only happen once due to the boolean `allocatorTokensToBeReturned[_allocator]` being set to `false` at the end of the call.

```
    function markUnallocatedTokensReturned(address _allocator, Types.AllocatorType
_allocatorType, uint256 _tokensReturned) external {
        require(msg.sender == owner() || allocatorTokensToBeReturned[msg.sender],
"Venture: Caller not a venture allocator");
        ...
        allocatorTokensToBeReturned[_allocator] = false;
    }
```

This means that only one call to either `close` or `terminateVesting` can happen per allocator contract.

One way to route around this is to add a new allocator with `_tokensForAllocation = 0` inside the `Venture` contract each time `close` or `termianteVesting` is called. This is because the `Venture.addAllocator` function sets the `allocatorTokensToBeReturned` mapping value back to `true`. However, this is not practical.

Impact

Either the `close` or `terminateVesting` functions can only be called once, unless `Venture.addAllocator` is called before every subsequent call.

Recommendation

In `Venture.markUnallocatedTokensReturned`, include an exception for option allocators, such that the `allocatorTokensToBeReturned` mapping value is not set to `false` for options allocators.

In `Venture.markUnallocatedTokensReturned`:

```
    function markUnallocatedTokensReturned(address _allocator, Types.AllocatorType
_allocatorType, uint256 _tokensReturned) external {
        require(msg.sender == owner() || allocatorTokensToBeReturned[msg.sender],
"Venture: Caller not a venture allocator");
        ...
+       if (_allocatorType != Types.AllocatorType.ERC20_OPTIONS_ALLOCATOR) {
            allocatorTokensToBeReturned[_allocator] = false;
+       }
    }
```

Status

Resolved

[M-08] ERC20OptionsAllocator#allocateBulk - Venture owner can allocate unlimited amount of tokens

Description

In `allocateBulk`, the `remainingGlobalTokenAllocation` variable is incorrectly calculated. This allows the venture owner to allocate an unlimited amount of tokens.

Vulnerability Details

In `allocateBulk`, the tokens available to allocate is tracked through the `remainingGlobalTokenAllocation` variable.

```
        uint256 remainingGlobalTokenAllocation;
        for (uint256 i; i < accounts.length; ++i) {
            remainingGlobalTokenAllocation =
                totalTokensForAllocation -
                tokenAllocationAmounts[i];
            if (remainingGlobalTokenAllocation < tokenAllocationAmounts[i]) {
                revert(
                    "TokenAllocator: Insufficient tokens available for allocation"
                );
            }
            _allocate(tokenAllocationAmounts[i], accounts[i]);
        }
```

The value for `remainingGlobalTokenAllocation` is updated before every allocation as `totalTokensForAllocation - tokenAllocationAmounts[i]`. This calculation is incorrect.

As long as `totalTokensForAllocation >= 2 * tokenAllocationAmounts[i]`, the if-statement will pass. This means that as long as each individual allocation is less than half of the total tokens available, the venture owner can allocate as many tokens as they want, even if the contract has insufficient funds to pay them out.

Since the same account can appear twice in the `accounts` mapping, the venture owner can allocate as many tokens as they want.

Impact

The venture owner can allocate an unlimited amount of tokens to any account, even if the contract has insufficient funds to pay them out. This will result in an insolvency of the contract.

Recommendation

Two options:

1. Correct the calculation of `remainingGlobalTokenAllocation`:

```
-   uint256 remainingGlobalTokenAllocation;
+   uint256 remainingGlobalTokenAllocation = totalTokensForAllocation -
totalAllocationTokenAllocated;
    for (uint256 i; i < accounts.length; ++i) {
-       remainingGlobalTokenAllocation =
-           totalTokensForAllocation -
-           tokenAllocationAmounts[i];
-       if (remainingGlobalTokenAllocation < tokenAllocationAmounts[i]) {
-           revert(
-               "TokenAllocator: Insufficient tokens available for allocation"
-           );
-       }
+       remainingGlobalTokenAllocation -= tokenAllocationAmounts[i];
        _allocate(tokenAllocationAmounts[i], accounts[i]);
    }
```

   Keep in mind that there is no need to check that `remainingGlobalTokenAllocation` is non-negative as Solidity v0.8's checked math will handle that and revert if there's a negative amount of remaining tokens.

2. Instead of tracking `remainingGlobalTokenAllocation`, change the if-statement as shown below:

```
-   uint256 remainingGlobalTokenAllocation;
    for (uint256 i; i < accounts.length; ++i) {
-       remainingGlobalTokenAllocation =
-           totalTokensForAllocation -
-           tokenAllocationAmounts[i];
-       if (remainingGlobalTokenAllocation < tokenAllocationAmounts[i]) {
+       if (totalAllocationTokenAllocated >= totalTokensForAllocation) {
            revert(
                "TokenAllocator: Insufficient tokens available for allocation"
            );
        }
        _allocate(tokenAllocationAmounts[i], accounts[i]);
    }
```

   This option is slightly simpler than the first, but costs more gas since the `totalAllocationTokenAllocated` storage variable will need to be accessed for every allocation, costing 2,100 gas units for the first 'cold access', and 100 gas units for every subsequent 'warm accesses'.

We recommend the first option, as it’s a lot more gas efficient.

Status

Resolved

Low

[L-01] Missing non-zero address checks in the constructor

Description

The constructors of contracts are missing zero address checks to ensure that addresses are initialized properly.

Recommendation

Add non-zero address checks for all address type arguments.

Status

Acknowledged

> JubiDAO has acknowledged this issue but have decided to not address it since it is low priority.

[L-02] Venture - `signatureStore` mapping is not used

Description

The `signatureStore` mapping inside `Venture` is defined but not used.

Recommendation

Remove the mapping.

Status

Resolved

[L-03] Venture - Duplicate lines of code

Description

There are several instances where code has been duplicated.

```
using SafeERC20 for IERC20;
using SafeERC20 for IERC20;
...

function addAllocator(address _newAllocator, Types.AllocatorType _allocatorType,
uint256 _tokensForAllocation) external  {
    require(_isAdminOrAllocatorManager(tx.origin), "Venture: Only Admin or Manager");
    allocators.push(_newAllocator);
    allocatorType[_newAllocator] = _allocatorType;
    if (_allocatorType == Types.AllocatorType.ERC20_FIXED_PRICE) {
        totalTokensAllocated[_newAllocator] = totalTokensAllocated[_newAllocator] +
_tokensForAllocation;
    }
    if (_allocatorType == Types.AllocatorType.NFT) {
        totalTokensAllocated[_newAllocator] = totalTokensAllocated[_newAllocator] +
_tokensForAllocation;
    }
```

```
    emit AllocatorAdded(_newAllocator, _allocatorType, _tokensForAllocation);
}

//return unallocated tokens from an allocator and decrease totalAllocated for the
token
function markUnallocatedTokensReturned(address _allocator, Types.AllocatorType
_allocatorType, uint256 _tokensReturned) external {
    require(_isAdminOrAllocatorManager(tx.origin), "Venture: Only Admin or Manager");
    if (_allocatorType == Types.AllocatorType.ERC20_FIXED_PRICE) {
        totalTokensAllocated[_allocator] = totalTokensAllocated[_allocator] -
_tokensReturned;
    }
    if (allocatorType[_allocator] == Types.AllocatorType.NFT) {
        totalTokensAllocated[_allocator] = totalTokensAllocated[_allocator] -
_tokensReturned;
    }
    emit unallocatedTokensReturned(_allocator, _allocatorType, _tokensReturned);
}
...
```

Recommendation

We suggest if totalTokensAllocated is the same in both checks, so remove if statements.

Status

Resolved

[L-04] Venture - No need to grant unused roles

Description

In contract initializers there are grants `DEFAULT_ADMIN_ROLE` to multiple addresses, and revoke roles at same time for initializers.

```
function initialize(Types.VentureConfig memory config, address _creator) external
initializer {
    // Ownable
    _transferOwnership(config.fundsAddress);

    // AccessControl
    _grantRole(DEFAULT_ADMIN_ROLE, config.fundsAddress);
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _grantRole(DEFAULT_ADMIN_ROLE, _creator);
    ...
    _revokeRole(ADMIN, msg.sender);
    _revokeRole(DEFAULT_ADMIN_ROLE, msg.sender);
    ...
}
```

Recommendation

We suggest removing unused grant roles functionality, there is no need to grant unused roles.

Status

Acknowledged

> JubiDAO has decided to not implement a fix since the issue only affects gas costs on deployment.

[L-05] Spelling mistakes

Description

There are multiple instances of spelling mistakes in comments:

`ERC20FixedPriceAllocator`

"sufficent" should be "sufficient". "purchasor" should be "purchaser". "happes" should be "happens". "purcased" should be "purchased".

`Factory`

"varaibles" should be "variables".

`Venture` "oldAmin" should be "oldAdmin".

Recommendation

Fix these spelling mistakes.

Status

Resolved

[L-06] ERC20FixedPriceAllocator - No need to set same value

Description

The variable `isOpen` is assigned as `true` in its definition. This is unnecessary, since the use of proxies means that this assignment will not be reflected in the proxy contract. Furthermore, the `initialize` function already sets the value of `isOpen` to `true`.

Recommendation

We suggest removing the default value `true` from the `isOpen` variable as the `isOpen` variable will become `true` when the contract is initialised.

Status

Resolved

[L-07] Allocator Contracts - Allocated tokens check can potentially be side-stepped

Description

The following condition in `_allocate` exists in `ERC20ManualAllocator` and `ERC20OptionsAllocator`:

```
    if (totalAllocationTokenAllocated == totalTokensForAllocation) {
            _close();
    }
```

This condition is used to check if all tokens have been allocated. However, it can potentially be side-stepped if an exploit allows the venture admin to allocate more tokens than `totalTokensForAllocation`.

Other conditions where the same issue holds are below.

In `ERC20OptionsAllocator.executeOption`:

```
    if (
        allocation[msg.sender] == accountOptionsExercised[msg.sender]
    )
```

In `ERC20ManualAllocator.calculateReleased`:

```
    if (
        totalAllocationTokenAllocated == totalClaimed ||
        allocation[account] == accountClaimed[account]
    ) {
        revert("Allocator: All tokens have been claimed");
    }
```

Recommendation

Change the `==` to `>=` in these types of occurrences.

Status

Acknowledged

[L-08] JubiERC20BasicImpl#burn - Function can only be called by minters

Description

There is no indication in the Natspec comments that the `burn` function can only be called by minters. However, the `onlyMinter` modifier is used.

Recommendation

Depending on intended behavior, either remove the `onlyMinter` modifier or update the Natspec comment for the `burn` function.

Status

Resolved

[L-09] JubiERC20BasicImpl#mint - Function can be paused

Description

There is no indication in the Natspec comments that the `mint` function is able to be paused. However, the `_mint` function contains the `_beforeTokenTransfer` hook which checks for the paused state.

In `JubiERC20BasicImpl`:

```
    function _beforeTokenTransfer(
            address from,
            address to,
            uint256 amount
        ) internal override(ERC20) {
            // If paused, the owner can still transfer their tokens.
            require(!transferPaused || msg.sender == owner(), "Pausable: paused");

            super._beforeTokenTransfer(from, to, amount);
    }
```

In `ERC20`:

```
    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer(address(0), account, amount);
        ...
    }
```

Recommendation

Depending on intended behavior, either override the `_mint` function to not include the `_beforeTokenTransfer` hook or update the Natspec comment for the `mint` function.

Status

Resolved

[L-10] TokenCreator Contracts - Missing input validation for owner address

Description

The `createToken` function inside the TokenCreator contracts is missing input validation for `_config.owner`. This means that `_config.owner` can be the zero address, which will result in there being no owner for the token contract.

Recommendation

Add input validation to `createToken` to make sure that `_config.owner` is not the zero address.

```
    require(_config.owner != address(0), "TokenCreator: Owner is zero address");
```

Status

Resolved

[L-11] ERC20ManualAllocator - `tokenPrice` state variable serves no purpose

Description

The `tokenPrice` state variable is not initialized or used anywhere in the contract. This makes sense since the contract is used for distributing venture tokens only and not for funds exchange.

Recommendation

Remove the `tokenPrice` state variable.

Status

Acknowledged

> JubiDAO has stated that the purpose of the state variable is external to the business logic of the contract.

[L-12] Allocator Contracts - Contract is upgradeable but does not inherit from upgradeable contracts

Description

The `ERC20ManualAllocator` contract will be interacted with via a proxy as indicated by the use of the `initialize` function. However, it uses `Ownable` and `ReentrancyGuard` which are not upgradeable contracts.

This also applies to `ERC20OptionsAllocator` and `ERC20FixedPriceAllocator`, which have the same inheritance pattern.

Vulnerability Details

When interacting with contracts via proxy, the constructors of the logic contracts are replaced by an `initialize` function, so that the proxy contract can use `delegatecall` to initialize state variables in the context of its own storage, as opposed to the storage of the logic contract. The same also has to apply to parent contracts that the logic contract inherits from.

Upgradeable variants of contracts that replace the constructors with `initialize` functions need to be used. However, the `Ownable` and `ReentrancyGuard` contracts which `ERC20ManualAllocator` inherits from are not upgradeable.

For `Ownable`, there is no impact as `_transferOwnership` is called inside `initialize`, which transfers the ownership from the zero address (as a result of `Ownable` not being initialized) to the venture owner.

For `ReentrancyGuard`, the reentrancy `_status` variable will not be initialized and will be set to `0` by default.

In `ReentrancyGuard`:

```
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2

    uint256 private _status;

    constructor() {
        _status = _NOT_ENTERED;
    }

    ...

    modifier nonReentrant() {
        _nonReentrantBefore();
```

```
        _;
        _nonReentrantAfter();
    }

    function _nonReentrantBefore() private {
        // On the first call to nonReentrant, _status will be _NOT_ENTERED
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

        // Any calls to nonReentrant after this point will fail
        _status = _ENTERED;
    }

    function _nonReentrantAfter() private {
        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
        _status = _NOT_ENTERED;
    }
```

The contract uses constants `1` and `2` to denote the `_NOT_ENTERED` and `_ENTERED` status. Fortunately, the `_nonReentrantBefore` function checks for `_status != _ENTERED`, so there is no impact to the contract's functionality.

Recommendation

Even though there is no tangible impact, it's recommended to follow best practices and use upgradeable variants of contracts when using upgradeable contracts.

Inherit from `OwnableUpgradeable` and `ReentrancyGuardUpgradeable` contracts and initialize them inside the `initialize` function.

Don't forget to change the order of inheritance so that the most base-like contract gets inherited first (`Initializable` gets inherited first).

In `ERC20ManualAllocator`:

```
- contract ERC20ManualAllocator is Ownable, Initializable, ReentrancyGuard {
+ contract ERC20ManualAllocator is Initializable, OwnableUpgradeable,
ReentrancyGuardUpgradeable {
     function initialize(
        Types.AllocatorConfig memory _config
    ) external initializer {
+       __Ownable_init();
+       __ReentrancyGuard_init();
        _transferOwnership(_config.venture.owner());
        ...
    }
```

The same fix applies to `ERC20FixedPriceAllocator` and `ERC20ManualAllocator`.

Status

Resolved

[L-13] Option & Fixed Price Allocators - Incorrect cost calculation due to inconsistent token decimals

Description

The `allocationToPurchaseToken` function does not take into account different token decimals. It assumes that the `allocationToken` has 18 decimals, and does not indicate and take into account the decimals of `strikePrice` or `tokenPrice`.

Vulnerability Details

The `allocationToPurchaseToken` function is used to calculate the amount of `purchaseToken` that the user has to send to the `venture` contract to purchase `allocationToken`.

In `ERC20OptionsAllocator`:

```
    function allocationToPurchaseToken(
        uint256 _allocation
    ) internal view returns (uint256) {
        return (_allocation / 1e18) * strikePrice;
    }
```

In `ERC20FixedPriceAllocator`:

```
    function allocationToPurchaseToken(
        uint256 _allocation
    ) internal view returns (uint256) {
        // TODO, will this result in USDC decmimals?
        return (_allocation / 1e18) * tokenPrice;
    }
```

Because the allocators assume that decimals coming through are 1e18, this may cause a mathematical error should the token be of any other decimals, which can result in the attacker being able to claim more or less than they're entitled to. Consider ERC20 tokens such as USDC or USDT which are of 1e6. Note that this function is used against both venture and treasury tokens for `ERC20FixedPriceAllocator`.

This may also impact the Jubi fees in `ERC20FixedPriceAllocator` as well. The contract might overflow and revert during the purchase.

The function also does not indicate and take into account the decimals of `strikePrice` or `tokenPrice`. This will result in an incorrect cost calculation if the `strikePrice` or `tokenPrice` decimals deviate from the decimals of `purchaseToken`.

Recommendation

To account for `allocationToken` decimals, divide `_allocation` by `10 ** allocationToken.decimals()` instead of `1e18`. Keep in mind that the `decimals` method is not available in the `IERC20` interface. Use the `IERC20Metadata` interface instead.

To account for `strikePrice` and `purchaseToken` decimals, there are three options:

1. Allow the venture admin to set the number of decimals for the `strikePrice` or `tokenPrice` at initialization. Create a `Price` struct that includes the `strikePrice` or

`tokenPrice`, as well as a `priceDecimals` field to indicate the number of decimals for the `strikePrice`. This struct can then be used in the `allocationToPurchaseToken` function. This comes at the cost of complexity and higher gas costs.

2. Set a fixed number of decimals for `strikePrice` or `tokenPrice` and make it clear in Natspec comments and documentations that venture admins must adhere to this standard. A good fixed number is 18 decimals.

3. Use the same number of decimals for `strikePrice` or `tokenPrice` as `purchaseToken`. This is the most gas efficient method but is prone to human error.

Combined with the recommended fix for [H-01] the `ERC20OptionsAllocator` contract should look like the following (Option 1 was chosen):

```
    Price strikePrice;

    ...

    struct Price {
        uint256 price;
        uint256 priceDecimals;
    }

    ...

    function initialize(
        Types.AllocatorConfig memory _config
    ) external initializer {
        ...
        strikePrice.price = _config.tokenPrice;
        strikePrice.priceDecimals = _config.tokenPriceDecimals;
        ...
    }

    ...

    function allocationToPurchaseToken(
        uint256 _allocation
    ) internal view returns (uint256) {
        return
            (
                _allocation *
                strikePrice.price *
                (10 ** purchaseToken.decimals())
            ) /
            (
                (10 ** allocationToken.decimals()) *
                (10 ** strikePrice.decimals)
            );
    }
```

The same fix applies to `ERC20FixedPriceAllocator`.

Status

Resolved

Gas

[G-01] ERC20ManualAllocator - `isOpen` is initialized to `true` in the logic contract

Description

The `isOpen` variable is initialized to `true` in the logic contract.

```
    bool public isOpen = true;
```

This is unnecessary, as the proxy contract will initialize the variable when delegate-calling the `initialize` function.

Recommendation

Remove the initialization of `isOpen` in the logic contract.

```
-   bool public isOpen = true;
+   bool public isOpen;
```

Status

Resolved

[G-02] Prioritise more common `require` checks

Description

There are functions that use multiple `require` checks. For these cases, it's gas efficient to put checks that are more likely to fail first.

Recommendation

Checks that are more likely to fail more than others should be placed first.

Status

Resolved

[G-03] Cache to local variables for repeated state variable access

Description

There are instances where a state variable is accessed and used multiple times inside one function scope. This is gas inefficient.

Recommendation

Cache the state variable to memory and use that variable instead.

Status

Resolved

[G-04] No need to initialise variables with default values

Description

Any variable not explicitly set or initialised assumes a default value (0 for uint, false for bool, address(0) for address, and so forth). Explicitly initialising these variables with their default values is unnecessarily consuming gas.

Recommendation

If this initialization was performed for the sake of clarity, use a comment to indicate the default state of the variable.

For example, you can add a comment like

```
// uint i starts at zero
```

to illustrate that the variable `i` begins with a default value of zero.

Status

Resolved

[G-05] Revert strings longer than 32 bytes cost extra gas

Description

Each extra memory word of bytes past the original 32 incurs an `MSTORE` which costs 3 gas.

Recommendation

Use custom errors instead of using revert strings.

Status

Acknowledged

> JubiDAO has acknowledged the gas optimization finding and have decided to not implement a change due to time constraints.

[G-06] TokenCreator Contracts - Unnecessary check for `_config.minterburner` array length

Description

The `createToken` function inside TokenCreator contracts has a check for the length of the `_config.minterburners` array to be non-zero (`length != 0`).

This is unnecessary, as the for-loop inside the if-statement can account for situations where `length = 0`. If `length = 0`, the code inside the for-loop will not run and no minters will be set.

Recommendation

Remove the if-statement.

Status

Acknowledged

> JubiDAO has acknowledged the gas optimization finding and have decided to not implement a change due to time constraints.

[G-07] ERC20OptionsAllocator#executeOption - Checking for zero allocation is unnecessarily done multiple times

Description

When a user calls `executeOption`, the following if-statement is checked three times:

```
    if (allocation[msg.sender] == 0){
        revert(
            "Allocator: Account not allocated tokens"
        );
    }
```

This is because the same if-statement is inside `executeOption`, `calculateVested` and `calculateVestedAt`. This is redundant.

Recommendation

Include the if-statement inside `executeOption` only. It is unnecessary in the `view` functions.

Status

Acknowledged

> JubiDAO has acknowledged the gas optimization finding and have decided to not implement a change due to time constraints.

[G-08] Combine similar if checks

Description

There are functions that have two if statements that check for different conditions but execute similar logic. These two checks could be combined into a single if statement.

```
function addAllocator(address _newAllocator, Types.AllocatorType _allocatorType,
uint256 _tokensForAllocation) external  {
    require(_isAdminOrAllocatorManager(tx.origin), "Venture: Only Admin or Manager");
    allocators.push(_newAllocator);
    allocatorType[_newAllocator] = _allocatorType;
    if (_allocatorType == Types.AllocatorType.ERC20_FIXED_PRICE) {
        totalTokensAllocated[_newAllocator] = totalTokensAllocated[_newAllocator] +
_tokensForAllocation;
    }
    if (_allocatorType == Types.AllocatorType.NFT) {
        totalTokensAllocated[_newAllocator] = totalTokensAllocated[_newAllocator] +
_tokensForAllocation;
    }
    emit AllocatorAdded(_newAllocator, _allocatorType, _tokensForAllocation);
}
```

```
//return unallocated tokens from an allocator and decrease totalAllocated for the
token
function markUnallocatedTokensReturned(address _allocator, Types.AllocatorType
_allocatorType, uint256 _tokensReturned) external {
    require(_isAdminOrAllocatorManager(tx.origin), "Venture: Only Admin or Manager");
    if (_allocatorType == Types.AllocatorType.ERC20_FIXED_PRICE) {
        totalTokensAllocated[_allocator] = totalTokensAllocated[_allocator] -
_tokensReturned;
    }
    if (allocatorType[_allocator] == Types.AllocatorType.NFT) {
        totalTokensAllocated[_allocator] = totalTokensAllocated[_allocator] -
_tokensReturned;
    }
    emit unallocatedTokensReturned(_allocator, _allocatorType, _tokensReturned);
}
```

Recommendation

Combine these into a single if statement.

Status

Resolved

Centralisation

The project is sufficiently decentralised without sacrificing security.

The owner can update the factory contract that deploys allocator contracts.

However, ownership of the allocator contracts are transferred to the venture owner and cannot be upgraded after the allocator contract has been deployed.

The factory contract retains the ability to attack a new allocator to a venture.

Conclusion

After Hashlocks analysis, the Jubi Dao project seems to have a sound and well tested code base. Overall, the majority of the code is correctly ordered and follows industry best practices. The code is very well commented as well.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits are to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and whitebox penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contracts details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds, and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3 oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au

#Hashlock.