BAIL security

Atlantis

FINAL REPORT

May ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Atlantis
Website	atlantisprotocol.so
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/atlantis-protocol/contracts/tree/712de11c93e1416268b02ba8c6afa287a798cbe3 * AtlantisFFLSale commit nor provided, only raw solidity file.
Resolution 1	https://github.com/atlantis-protocol/contracts-private/tree/f30041293399de8606475573013752d941bcc45e/contracts

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High	1	1			
Medium	5	4		1	
Low	2			2	
Informational	17	6		11	
Governance	3	1		2	
Total	28	12		16	

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium-level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless, the issue should be fixed immediately.
Informational	Effects are small and do not pose an immediate danger to the project or users.
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior.

3. Detection

AQUAToken

The AQUAToken is the primary utility token of the Atlantis Protocol.
It has a configurable pre-mine supply and a fixed maximum supply set at deployment.

Upon creation, the pre-mined amount is minted directly to the deployer (msg.sender). The maximum supply cannot be changed after deployment.

AQUAToken uses a minter system where only addresses added by the owner can mint new tokens, respecting the maximum supply constraint. Minters can also burn tokens from an account, provided they have been approved to do so through allowance.
Core Invariants:

INV 1: Minters cannot exceed the allowance by msg.sender in burnFrom()
INV 2: totalSupply cannot exceed the maxSupply
INV 3: Only whitelisted minters can call mint() and burnFrom()

Privileged Functions

- addMinter
- delMinter
- getMinter

Issue_01	Adding a malicious minter can inflate the totalSupply
Severity	**Governance**
Description	The owner can add minters using addMinter(). These minters can call mint(), to mint any amount of tokens, therefore inflating the totalSupply and decreasing the token price.
Recommendations	Consider undergoing a deep review or audits of the all minters' implementation before adding a new minter.
Comments / Resolution	Acknowledged.

Issue_02	Unnecessary onlyOwner modifier in getMinter()
Severity	**Informational**
Description	getMinter() is only a view function, which does not perform any state changes, therefore it is unnecessary to guard it via the onlyOwner modifier.
Recommendations	Consider removing the onlyOwner modifier from getMinter() in both AQUAToken and xAQUAToken.
Comments / Resolution	Resolved by following the recommendation.

xAQUAToken

The xAQUAToken is an escrowed token representing locked AQUAToken tokens.
It shares the same maximum supply as AQUAToken, enforced during minting. Users can lock AQUA to mint xAQUA at a 1:1 ratio, and redeem xAQUA to reclaim AQUA through authorized minters. Locked AQUA tokens are held by the xAQUAToken contract until redemption.

Minter management mirrors AQUA's system, with the owner controlling minter permissions. Transfers are restricted: either the sender or the recipient must be whitelisted to move xAQUA tokens, ensuring controlled circulation within approved addresses.

Core Invariants:

INV 1: Only whitelisted minters can call mint(), lock(), redeem() and burnFrom()
INV 2: A transfer requires the from or the to address to be whitelisted in _transferWhitelist
INV 3: Minters cannot exceed the allowance by msg.sender in burnFrom()
INV 4: totalSupply cannot exceed the maxSupply

Privileged Functions

- addMinter
- delMinter
- getMinter
- addToTransferWhitelist
- removeFromTransferWhitelist
- getTransferWhitelistAddress

Issue_03	Unnecessary onlyOwner modifier in getTransferWhitelistAddress()
Severity	**Informational**
Description	The getTransferWhitelistAddress() function returns the address of a whitelisted user based on the provided index from the _transferWhitelist EnumerableSet. It is a view-only function that does not modify the contract state, so it is unnecessarily restricted by the onlyOwner modifier.
Recommendations	Consider removing the onlyOwner modifier from getTransferWhitelistAddress().
Comments / Resolution	Resolved by following the recommendation.

Issue_04	Unnecessary maxSupply restriction on xAQUA totalSupply
Severity	**Informational**
Description	The _mint() function has performs the following restriction: *function _mint(address account, uint256 amount) internal virtual override {* *require(totalSupply().add(amount) <= maxSupply, "xAQUA: mint amount exceeds max supply");* *super._mint(account, amount);* *}* However, it is impossible to reach the *xAQUA: mint amount exceeds max supply* error, because the AQUA.mint() function is invoked prior to this check: *function mint(address _to, uint256 _amount) public onlyMinter returns (bool) {* *AQUA.mint(address(this), _amount);* *_mint(_to, _amount);* *return true;*

	} Since AQUAToken::mint() will also perform the totalSupply validation on the AQUA token, it is impossible for the xAQUA token totalSupply to go beyond the AQUA totalSupply.
Recommendations	Consider removing the redundant maxSupply logic from the xAQUAToken, as it is already performed in the AQUAToken.
Comments / Resolution	Resolved by not overriding the _mint() function.

Issue_05	Unused burnFrom() function
Severity	**Informational**
Description	The xAQUAToken contract implements burnFrom(). However, burning xAQUA tokens, without transferring out AQUA could lock AQUA tokens in the xAQUAToken contract. Furthermore, this function is not used by any of the contracts.
Recommendations	Consider removing the redundant burnFrom() function from xAQUAToken.sol.
Comments / Resolution	Resolved by following the recommendation.

AtlantisStaking

The AtlantisStaking contract connects the AQUA and xAQUA tokens, enabling flexible staking with reward incentives.
Users can lock tokens for 15 to 90 days, with reward ratios scaling from 50% to 100% based on the lock duration.
The staking follows a two-step redemption process: users first initialize a redemption by locking xAQUA for a chosen period (initializeRedeem()), and then finalize it after the unlock time to claim their rewards (finalizeRedeem()).
The system tracks user lock positions through a structured LockInfo, recording the locked amount, unlock time, reward amount, and lock duration.
Redeeming early is prevented by strict time checks. After the lock expires, only the calculated reward is transferred back, and any residual AQUA is burned.

Appendix: Redeem Ratio

The redeem ratio is calculated using a linear formula that adjusts proportionally based on the selected lock duration in days:

$$ratio = minRedeemRatio + \frac{(days - minRedeemDays) * (maxRedeemRatio - minRedeemRatio)}{maxRedeemDays - minRedeemDays}$$

- ➔ minRedeemDays - the minimum duration of a stake
- ➔ minRedeemRatio - the redeem ratio for minRedeemDays
- ➔ maxRedeemDays - the maximum duration of a stake
- ➔ maxRedeemRatio - the redeem ratio for maxRedeemDays

Core Invariants:

INV 1: Users cannot call finalizeRedeem() before the unlockTime has passed
INV 2: initializeRedeem() cannot have a lock duration longer than the maxRedeemDays
INV 3: initializeRedeem() cannot have a lock duration shorter than the minRedeemDays

Privileged Functions

- None.

Issue_06	finalizeRedeem() will always revert due to exceeding the allowance
Severity	High
Description	In finalizeRedeem(), any residual AQUA tokens are burned: *if (AQUAresidual > 0) {* *// No need to approve ourselves* *token.burnFrom(address(this), AQUAresidual);* *}* No approval was granted since the msg.sender for the burnFrom() function is also the spender of the AQUA tokens. However, in the AQUAToken contract, approval is in fact required, even though the spender is the same as the sender: *function burnFrom(address account, uint256 amount) external onlyMinter {* *uint256 currentAllowance = allowance(account, _msgSender());* *require(* *currentAllowance >= amount,* *"AQUA: burn amount exceeds allowance"* *);* *_approve(account, _msgSender(), currentAllowance.sub(amount));* *_burn(account, amount);* Here, the allowance() function is called to fetch the allowes from the AtlantisStaking contract to itself: */// @inheritdoc IERC20* *function allowance(address owner, address spender) public view virtual returns (uint256) {* *return _allowances[owner][spender];* *}* Since no such approval was made and allowance will be 0, the

	transaction will always revert, causing a DOS of finalizeRedeem() and preventing users from claiming their AQUA tokens.
Recommendations	Consider skipping the allowance check in burnFrom() when the *msg.sender == account*.
Comments / Resolution	Resolved by adding an approval before burning.

Issue_07	Users may receive less tokens due to rounding in getRedeemRatio()
Severity	**Medium**
Description	The following formula is used to determine the redeem ratio: *function getRedeemRatio(uint256 _days) public view returns (uint256) {* *if (_days >= maxRedeemDays) return maxRedeemRatio;* *else if (_days == minRedeemDays) return minRedeemRatio;* *return* *minRedeemRatio.add(* *(_days.sub(minRedeemDays))* *.mul(maxRedeemRatio.sub(minRedeemRatio))* *.div(maxRedeemDays.sub(minRedeemDays))* *);* However, the redeemRatio lacks precision, meaning it only supports whole number percentage values and cannot accommodate fractional percentages: *uint256 rewardAmount = _amount.mul(ratio).div(100);* As a result of this lack of precision, users may not receive additional tokens even if they opt for a slightly longer lock duration. Consider the following scenario:

	minRedeemDays = 15 days maxRedeemDays = 90 days minRedeemRatio = 50 (50%) maxRedeemRatio = 100 (100%) Case 1: (_days = 15 - minimum) → For _days = 15, the minRedeemRatio is applied: *if (_days == minRedeemDays) return minRedeemRatio;* So it will return exactly 50 (50%). Case 2: (_days = 16) → For _days = 16, the following formula is used: *minRedeemRatio.add(* *(_days.sub(minRedeemDays))* *.mul(maxRedeemRatio.sub(minRedeemRatio))* *.div(maxRedeemDays.sub(minRedeemDays))* *)* Due to rounding the result would be: $$50 + \frac{(16 - 15) * (100 - 50)}{90 - 15} = 50 + \frac{50}{75} = 50$$ This means the reward ratio will be the same for users who lock tokens for 15 days and users with a one day longer lock.
Recommendations	Consider using an increased precision for the redeemRate calculation.
Comments / Resolution	Resolved by adding increased precision.

Issue_08	Unused prevReward struct variable in the LockInfo
Severity	**Informational**
Description	The LockInfo contains the following variables: *struct LockInfo {* *uint256 amount;* *uint256 prevReward;* *uint256 unlockTime;* *uint256 day;* *uint256 rewardAmount;* *}* However, everytime the struct is used the prevReward is always hardcoded to 0: *LockInfo memory info = LockInfo(_amount, 0, unlockTime, _days, rewardAmount);* As a result prevReward stays unused.
Recommendations	Consider removing prevReward from LockInfo.
Comments / Resolution	Resolved by removing prevReward.

Issue_09	CEI Violation In Finalize Redeem
Severity	**Informational**
Description	In the finalizeRedeem() function the token transfer is performed before redeem entry is deleted → *if (AQUAresidual > 0) {* *// No need to approve ourselves* *token.burnFrom(address(this), AQUAresidual);* *}* *token.transfer(msg.sender, info.rewardAmount);* *_deleteRedeemEntry(index, msg.sender);* *}* Currently the token can be only AQUA token , but if in future the token is any token including ERC-777 token then using a post transfer hook the user can reenter the function to get more rewards since the redeem entry is deleted after the transfer.
Recommendations	Consider transferring the tokens after deleting the redeem entry.
Comments / Resolution	Resolved by following the recommendation.

Issue_10	_beforeTokenTransfer unnecessarily calls the parent function
Severity	Informational
Description	The override of _beforeTokenTransfer unnecessarily calls the parent implementation even though it is a no-op. *super._beforeTokenTransfer(from, to, amount);*
Recommendations	Consider removing the call.
Comments / Resolution	Acknowledged.

AtlantisFFLSale

The AtlantisFFLSale contract handles the Fair Farming Launch and the initial contributions of users by managing a time-restricted ETH-based token sale involving multiple ERC20 tokens. It supports configurable minimum and maximum contributions per user, referral-based ETH forwarding (measured in basis points), and an optional burn mechanism that conditionally reduces token distribution if the burnThreshold is not met. The contract also supports optional reward token emissions, distributed proportionally based on SONIC contributions over time. Token accounting accounts for fee-on-transfer tokens by measuring balance deltas. SONIC contributions may be forwarded to a specified vault address or retained by the contract. After the sale ends—either naturally or manually—users can claim their proportional share of sale tokens and rewards(claimTokens()).

Appendix: Emission Calculation

The contract updates the emissionPerETH value based on elapsed time and total SONIC raised, distributing the configured reward token linearly over the sale period. The computation follows these steps:

1) Determine the current applicable time:

$$currentTime = Min(block.timestamp, endTime)$$

2) Determine the effective last update time:

$$effectiveLastUpdate = Max(rewardLastUpdate, startTime)$$

3) Calculate the time elapsed since the last reward update:

$$timeElapsed = currentTime - effectiveLastUpdate$$

4) Calculate the total sale duration:

$$totalSaleDuration = endTime - startTime$$

5) Calculate the proportional reward emission:

$$emission = \frac{totalRewardAmount * timeElapsed}{totalSaleDuration}$$

6) Update the emission accumulator per unit of ETH contributed:

$$emissionPerETH \mathrel{+}= (emission * 1e18) \;/\; totalRaised$$

➔ totalRaised - the total amount currently raised by contributors

Appendix: Reward Distribution

Once the global emissionPerETH value is updated to reflect new emissions, the individual user's reward state must also be updated to account for their new contribution. This is done in two steps:

1) **Calculate the user's pending rewards accrued before the current contribution:**

$$pendingRewards = \frac{prevContributionAmount * emissionPerEth}{1e18} - missedRewardEmissions$$

➔ prevContributionAmount - amount before adding the new contribution.

2) **Update the user's missed rewards based on the new contribution total:**

$$missedRewardEmissions = \frac{(newTotalContribution * emissionPerETH)}{1e18} - pendingRewards$$

➔ newTotalContribution - the total amount contributed by the user after the new contribution was added.

Appendix: Distribution Ratio

If the total amount of SONIC raised during the sale does not meet the configured burnThreshold, contributors receive only a proportional share of the allocated sale tokens. The remaining undistributed portion is permanently removed from circulation—either by calling the token's burn() function (if implemented) or by transferring the excess tokens to the predefined BURN_ADDRESS (0x000...dEaD).

The proportion of tokens to be distributed to users is determined by the distributionRatio, calculated as:

$$distributionRatio = (totalRaised * 1e18) \;/\; burnThreshold$$

Core Invariants:

INV 1: Contributions occur after a sale has started.
INV 2:Contributions cannot be done after the endTime of a sale and before the startTime.
INV 3: contributionAmount shall be more than the minContribution.
INV 4: newTotalContribution cannot exceed the maxContribution.
INV 5: Purchased tokens and rewards can only be claimed after the sale ends.
INV 6: Owner can add sale tokens up to the MAX_SALE_TOKENS.
INV 7: Owner cannot add reward tokens after sale has started or reward has been set.
INV 8: updateSaleTimes() cannot change the startTime if sale has already started.
INV 9: updateSaleTimes() cannot change endTime to be earlier than previously set.

Privileged Functions

- addToken
- setRewardToken
- setVaultAddress
- startSale
- endSale
- updateSaleTimes
- pause
- unpause
- emergencyWithdrawTokens
- emergencyWithdrawETH

Issue_11	endTime can be extended indefinitely
Severity	**Governance**
Description	The owner can call updateSaleTimes() to postpone the endTime. However, there are no restrictions allowing the owner to set the endTime for any future time, causing users to be unable to claim their tokens until the time has passed.
Recommendations	Consider enforcing a restriction on the endTime change.
Comments / Resolution	Resolved by removing updateSaleTimes().

Issue_12	Owner can withdraw all tokens
Severity	**Governance**
Description	The owner can call emergencyWithdrawTokens() and emergencyWithdrawETH() to withdraw all the user's assets from the contract anytime.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_13	updateSaleTimes() may cause reward insolvency
Severity	Medium
Description	The updateSaleTimes() function can be used to postpone the endTime during an active sale: *function updateSaleTimes(* *uint256 _startTime,* *uint256 _endTime* *) external onlyOwner {* *...* *} else {* *// If sale has started, cannot change start time.* *require(* *_startTime == startTime,* *"AtlantisFFLSale: Cannot change start"* *);* *// New end time must be after the current one (can only extend)* *require(_endTime > endTime, "AtlantisFFLSale: End must be later");* *endTime = _endTime;* *}* However, changing the endTime will cause accounting issues for the reward distribution as more rewards will be required. Consider the following scenario: 1) rewardTokenConfig.totalAmount = 50e18 startTime = 0 endTime = 30 2) At timestamp = 15 a contribution occurs, which will invoke _updateEmission(). As a result, an emission of **25e18 ($\frac{50e18 * 15}{30}$)** tokens will be accounted for in the emissionPerETH. 3) At timestamp = 15, the owner will also call updateSaleTimes(), changing the endTime to 40. 4) At timestamp = 40, the last contribution occurs, which calculates an emission of:

	$\frac{(40-15)*50e18}{40} = \mathbf{31.25e18}$ which is also distributed through emissionPerETH. In this example, instead of distributing the 50e18 (initially allocated as rewards), a total amount of 25e18+31.25e18 = **56.25e18** will be allocated for distribution, leading to reward insolvency. As a result, due to the limitation the amountToSend: *uint256 amountToSend = Math.min(rewardAmount, availableReward);* *if (amountToSend > 0) {* *rewardTokenConfig.token.safeTransfer(* *msg.sender,* *amountToSend* *);* The last user to call claimTokens() will unfairly receive a decreased reward.
Recommendations	Consider using a fixed endTime for the reward distribution, so that extending the sales does not cause an increase of rewards. Alternatively, the formula could be refactored to properly adjust when postponing the endTime, changing the reward rate.
Comments / Resolution	Resolved by removing updateSaleTimes().

Issue_14	Precision truncation could allow reward griefing for low decimal tokens
Severity	**Medium**
Description	The following formula is used to calculate the emissionPerETH: *emissionPerETH += (emission * 1e18) / totalRaised;* The emission is scaled using the rewardToken decimals, while the totalRaised will be in 18 decimals. Consider the following scenario: totalAmount = 5000e6 USDC totalRaised = 300_000e18 timeElapsed = 1 min totalSaleDuration = 15 days As a result the emissionPerEth will be rounded down to 0: $emission = \frac{5000e6 * 1min}{15*24*60} = 0.23e6$ $$emissionPerETH = \frac{0.23e6 * 1e18}{300000e18} = \frac{0.23e24}{0.3e24} = 0$$ This may be used by an attacker to grief the reward distribution by frequently depositing the minimum amount.
Recommendations	Consider implementing added precision for low decimal tokens.
Comments / Resolution	Acknowledged.

Issue_15	Less rewards will be distributed to users due to a missed update
Severity	Medium

Description	In claimTokens(), sale is automatically ended when the endTime is reached:
	
	function claimTokens() external nonReentrant {
	bool isTimeOver = block.timestamp > endTime;
	require(saleEnded || isTimeOver, "AtlantisFFLSale: Not ended");
	
	// If sale ended naturally via time passage, mark it internally
	if (!saleEnded && isTimeOver) {
	saleEnded = true;
	if (hasRewardToken) {
	_updateEmission(); // Final emission update using original endTime
	}
	
	However, due to setting saleEnded to true, prior to calling _updateEmission(), the last update will not be performed as the following condition will be entered:
	
	function _updateEmission() internal {
	// No update needed if no reward, not started, ended, or no time passed since last update
	if (
	!hasRewardToken ||
	!saleStarted ||
	saleEnded ||
	block.timestamp <= rewardLastUpdate
	) {
	return;
	}
	
	As a result, a portion of the rewards will not be distributed causing loss for users.
	Consider the following scenario:

	1) rewardTokenConfig.totalAmount = 50e18 startTime = 0 endTime = 30 2) At timestamp = 15, userA contributes, which will invoke _updateEmission(). As a result, an emission of **25e18** ($\frac{50e18 * 15}{30}$) tokens will be accounted for in the emissionPerETH(as missed for userA). 3) At timestamp = 40, no other contributions have occurred, so userA calls claimTokens(). Due to setting the saleEnded to true, _updateEmssion() will be returned early, which will cause the user to receive 0 rewards, instead of 25e18 (for the period from timestamp = 15 to 30).
Recommendations	Consider setting saleEnded to true after _updateEmissions() is invoked in claimTokens().
Comments / Resolution	Resolved by following the recommendation.

Issue_16	updateSaleTimes() should update the emission
Severity	**Medium**
Description	In updateSaleTimes(), the owner can postpone the endTime during an active sale: *function updateSaleTimes(* *uint256 _startTime,* *uint256 _endTime* *) external onlyOwner {* *...* *} else {* *// If sale has started, cannot change start time.* *require(* *_startTime == startTime,* *"AtlantisFFLSale: Cannot change start"* *);* *// New end time must be after the current one (can only extend)* *require(_endTime > endTime, "AtlantisFFLSale: End must be later");* *endTime = _endTime;* *}* However, changing the endTime will also change the reward emission rate, as the totalSaleDuration will be increased, causing a decrease in the reward rate. To ensure rewards are accurately updated, _updateEmission() must be called before modifying endTime. This prevents the new rate from being retroactively applied to the period between the last update and the updateSaleTimes() invocation.
Recommendations	Consider calling _updateEmission() prior to changing endTime in updateSaleTimes().
Comments / Resolution	Resolved by removing updateSaleTimes().

Issue_17	Flawed referral logic
Severity	**Low**
Description	In contribute(), users can specify a referral address to claim the referralPercent of their contribution: *function contribute(address referral) public payable nonReentrant whenNotPaused {* However, the only requirements for the referral address are the following: *if (referralPercent > 0 && referral != address(0) && referral != msg.sender) {* While ensuring that the referral is not the same as the msg.sender, to prevent self-referrals, users can still refer to themselves simply by providing an alternative wallet address that they own. As a result, they will use a discount for their contribution.
Recommendations	Consider implementing a referral whitelist, so that self-referrals are not feasible.
Comments / Resolution	Acknowledged.

Issue_18	Malicious referrer can gas-grief the user
Severity	Low
Description	A user can choose a referral address which will receive referral fee , from the UX side this would work as a bonus for users who refer other users and those users contribute in the sale contract. But a malicious referral address can gas grief a user , when the victim contributes using the contribute() function and sets a referral , this is how the referral fee is sent → *uint256 referralAmount = 0;* *if (* *referralPercent > 0 &&* *referral != address(0) &&* *referral != msg.sender* *) {* *referralAmount = (contributionAmount * referralPercent) / 10000;* *if (referralAmount > 0) {* *address payable referralPayable = payable(referral);* *(bool referralSuccess,) = referralPayable.call{* *value: referralAmount* *}("");* Therefore a malicious referral can consume “almost” all the gas such that the call is successful but the user is gas griefed.
Recommendations	Consider using a low-level call with a gas limit.
Comments / Resolution	Acknowledged.

Issue_19	Blacklisted users may be unable to claim anything
Severity	**Informational**
Description	In claimTokens(), the user is transferred amountsClaimed from all tokenAddresses. However, if one of the tokens is USDC and the user gets blacklisted, they will be unable to withdraw anything, due to a revert of the following transfer: *if (actualUserShare > 0) {* *config.token.safeTransfer(msg.sender, actualUserShare);* Some users may also be unable to claim if they have a pending rewardAmount of USDC tokens and become blacklisted.
Recommendations	Consider using try-catch for token transfers. Alternatively, we recommend acknowledging this issue if USDC is not to be used.
Comments / Resolution	Acknowledged.

Issue_20	Rewards may not be distributed in some cases
Severity	**Informational**
Description	Users can receive rewards in order to incentivize early participation. However, in the following cases, rewards may not be distributed: 1) A portion of the rewards will not be distributed for the period between the start of the sale and the first contribution. 2) Some rewards will not be distributed if the owner calls endSale() early. This will cause rewards between block.timestamp and the original endTime to remain undistributed. *Note: If this issue is fixed, so that all rewards are distributed, it may enable attackers to front-run the endSale() call and with a contribution and prevent full distribution of rewards.* 3) If there are no contributions, all rewards will remain undistributed.
Recommendations	As there are functions which allow the owner to withdraw tokens, we recommend acknowledging this issue if this behaviour is intended. *Note: Undistributed rewards shall be withdrawn cautiously, so that the owner does not accidentally withdraw pending unclaimed rewards, which belong to protocol's users.*
Comments / Resolution	Acknowledged.

Issue_21	claimTokens() could revert due to failed transfer
Severity	**Informational**
Description	claimTokens() will invoke _burnTokens() if there is any amountToBurn: *// Burn the non-distributed part first* *if (amountToBurn > 0) _burnTokens(config.token, amountToBurn);* In _burnTokens(), amountToProcess may be transferred to dead address if the token is not burnable: *// Fallback: Transfer to burn address.* *// Let it revert if transfer itself fails.* *_token.safeTransfer(BURN_ADDRESS, amountToProcess);* However, if this transfer fails, the whole claimTokens() call will be reverted, preventing users from claiming.
Recommendations	Consider using try-catch for the transfer to the BURN_ADDRESS.
Comments / Resolution	Acknowledged.

Issue_22	getTokenPrice() may return a wrong price
Severity	**Informational**
Description	When users call contribute(), a portion of the contribution is transferred to the referral address, but is not removed from the totalRaised. As a result, the amount allocated to referrers will be counted towards the token price: *uint256 priceBaseAmount = totalRaised;*
Recommendations	Consider if this behaviour is intended. If this is incorrect, consider discounting the referralAmount from the totalRaised.
Comments / Resolution	Acknowledged.

Issue_23	Contribution amount can go below minimum after accounting for referral fee
Severity	**Informational**

Description	Consider a user contributing 5 ETH and the minContribution has been set to 5 ETH , if the referralPercentage is say 10% then 0.5 ETH would be sent to the referral address and the actual contribution to the vault is 4.5 ETH which is less than the minContribution required and the check inside contribute() checks the amount without deducting the referral fee →
	
	UserInfo storage user = userInfo[msg.sender];
	uint256 contributionAmount = msg.value; // Cache msg.value
	uint256 newTotalContribution = user.totalContribution +
	contributionAmount;
	
	// Check contribution limits
	if (minContribution > 0) {
	require(

	contributionAmount >= minContribution, *"AtlantisFFLSale: Below min"* *};* *}*
Recommendations	Consider accounting for referral fee when verifying the contribution against minContribution.
Comments / Resolution	Acknowledged.

Issue_24	Incorrect accounting For FoT
Severity	**Informational**

Description	The AtlantisFFLSale supports fee-on-transfer tokens (reward tokens and claim tokens) , inside claimTokens() when reward amount is sent →
	
	if (rewardAmount > 0) {
	uint256 availableReward = rewardTokenConfig.token.balanceOf(
	address(this)
	);
	uint256 amountToSend = Math.min(rewardAmount, availableReward);
	if (amountToSend > 0) {
	rewardTokenConfig.token.safeTransfer(
	msg.sender,
	amountToSend
	); // Interaction
	}
	rewardAmount = amountToSend;
	
	However, it incorrectly assigns amountToSend as the rewardAmount since ***amountToSend - fee*** has been actually sent to

	the user. As a result, the following event may return wrong value: *emit TokensClaimed(* *msg.sender,* *tokensClaimed,* *amountsClaimed,* *rewardAmount* *);*
Recommendations	Consider accounting for the tokens actually transferred.
Comments / Resolution	Acknowledged.

Issue_25	Calling addToken after the sale ends will not allow all users to claim said token.
Severity	**Informational**
Description	The owner is permitted to call addToken which will add a token to the sale. The function does not include any restrictions, therefore the owner is able to call this function after the sale has ended. The problem occurs in the scenario when the token is added when the sale has ended and users have already claimed. In this scenario, users who have already claimed will not be able to claim any share of the newly added token because of the following check... *require(!user.hasClaimed, "AtlantisFFLSale: Already claimed");*
Recommendations	Consider not allowing tokens to be added after the sale has ended.
Comments / Resolution	Acknowledged.

Issue_26	Redundant call to _updateEmission
Severity	Informational

Description	In the claimTokens function, there are 2 instances of a call to _updateEmission. The first is near the beginning of the function.
	
	if (!saleEnded && isTimeOver) {
	if (hasRewardToken) {
	_updateEmission(); // Final emission update using original endTime
	}
	saleEnded = true;
	
	In the case where saleEnded has not been marked true but the isTimeOver function is true, we will call _updateEmission if hasRewardToken is true and then set saleEnded to true
	
	The next time _updateEmission is called happens later in the function. Keeping in mind that after the first snippet saleEnded will always be true, it is redundant to have another call to _updateEmission later in the function since this will result in a no-op due to the function returning early in case saleEnded is true
	
	function _updateEmission() internal {
	// No update needed if no reward, not started, ended, or no time passed since last update
	if (
	!hasRewardToken ||
	!saleStarted ||
	saleEnded ||
	block.timestamp <= rewardLastUpdate
	) {
	return;
	}

Recommendations	Consider acknowledging the issue or removing the call to _updateEmission.
Comments / Resolution	Acknowledged.

Issue_27	claimTokens does not have whenNotPaused modifier
Severity	**Informational**
Description	Users will be able to claim tokens during the paused state since the claimTokens function does not include the whenNotPaused modifier.
Recommendations	Consider adding the modifier or acknowledging the issue if this is desired behavior.
Comments / Resolution	Acknowledged.

Issue_28	Config should be read from memory
Severity	**Informational**
Description	Inside the for loop in the claimTokens function, the config variable is currently accessed from storage, despite being used as a read-only variable. Since it is not modified, copying it to memory would be more efficient and reduce gas costs. *TokenConfig storage config = tokenConfigs[tokenAddress];*
Recommendations	Consider acknowledging the issue or updating config to be memory to save gas. This issue can be safely acknowledged.
Comments / Resolution	Acknowledged.