BAIL security

Camelot
Grail

FINAL REPORT

June ‘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.

Bailsec has verified that the changes were solely cosmetic and did not introduce any logic changes.

Project	Camelot – Grail
Website	camelot.exchange
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/CamelotLabs/exchange-contracts/commit/45fa1a0a4afb9a87d6dab4c78a6af3f27c5e6292
Resolution 1	https://github.com/CamelotLabs/exchange-contracts/blob/d5d4115c4dc6d025ce08bbc6f9ac93bf569d404a/packages/grail/src/oracles/PriceFeed.sol
Resolution 2	https://github.com/CamelotLabs/exchange-contracts/tree/eb88733b90064566b328deda364e93cea4f1fb86/packages/grail/src NOTE: The naming scheme for the contract scope at the last commit has been changed This includes changing the following contract names: - GrailToken.sol to MainToken.sol - XGrailToken.sol to EscrowToken.sol as well as removal of all previous opinionated references to these tokens, contracts and interfaces from the function signatures and inputs, variables, events, errors, NatSpec and internal comments in the entire audit scope. Bailsec has verified that the changes were solely cosmetic and did not introduce any logic changes.

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High	2	2			
Medium	2	1		1	
Low	13	9		4	
Informational	16	6		10	
Governance					
Total	33	18		15	

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

PriceFeed

The PriceFeed contract fetches the price from the Algebra Volatility Oracle.

- Retrieves the tick for a specified period (which can be updated by the admin), then obtains the quote at that tick from the Volatility Oracle.

The getQuote() function takes an amount of baseToken to calculate the corresponding quotedToken amount for that input based on the price queried on the Algebra Volatility Oracle.

Core Invariants:

INV 1: returned value from getQuote represents the quote to get the specified amount of baseToken

INV 2: Queries to the VolatilityOracle can't be DoSed

Privileged Functions

- setVolatilityOracle
- setTimewindow

Issue_01	PriceFeed computes an incorrect quote for the specified amount of baseToken
Severity	**High**
Description	The tick of a pool (based on the TWAP) is being retrieved by calling getQuoteAtTick(). Then, the resulting rawQuote is normalized based on the decimal difference of the base and quote token. However, the last step is unnecessary and results in a completely incorrect quote amount. The tick of a token pair is simply a different form of the price of that token pair (specifically 1.0001^tick = price) where the price of a pool is, simplistically explained, how much token1 you would get for 1 token0 (with no price impact). By definition, this factors in decimal differences as for example a DAI/USDC with 1 token (with decimal precision) of liquidity each pool will have a price of 1e18 / 1e6 = 1e12. Thus, getQuoteAtTick() will return the quote token in quote token decimals. Thus, the decimal normalization in the if/else blocks actually makes the price incorrect, causing different issues like the user paying dust in OptionsToken::exercise(). The returned quote from VolatilityOracleInteractions.getQuoteAtTick() is scaled by extra decimals depending if baseTokenDecimals are > quoteTokenDecimals, but, the rawQuote is already scaled by the decimals of the quoteToken. Scaling up the rawQuote for tokens with different decimals will alter the real quote
Recommendations	It is not necessary to normalize the rawQuote returned from the VolatilityOracleInteractions.getQuoteAtTick()
Comments / Resolution	Fixed - Normalization logic has been removed, price is returned as it is queried from the VolatilityOracle

Issue_02	Decimal normalization in getQuote() is reversed
Severity	**Low**
Description	While this issue would usually be very significant, this code must be removed anyway as it is incorrectly added, thus it is marked as a Low. The following code is used for decimal normalization: *uint8 baseTokenDecimals = IERC20Metadata(baseToken).decimals();* *uint8 quoteTokenDecimals = IERC20Metadata(quoteToken).decimals();* *if (baseTokenDecimals > quoteTokenDecimals) {* *return rawQuote * (10 ** (baseTokenDecimals - quoteTokenDecimals));* *} else if (quoteTokenDecimals > baseTokenDecimals) {* *return rawQuote / (10 ** (quoteTokenDecimals - baseTokenDecimals));* *} else {* *return rawQuote;* *}* The target is to turn the base token decimals into quote token decimals. However, it happens incorrectly. Let's imagine baseTokenDecimals is 6 and quoteTokenDecimals is 18, thus we must increase the rawQuote by 12 decimals. We would end up in the quoteTokenDecimals > baseTokenDecimals block as 18 > 6 and we will divide rawQuote by 10 ** (18 - 6) = 1e12 which is incorrect, we must be multiplying in this case. Instead, this block would likely round down to 0 as we are dividing a 6-decimal token amount by 1e12.
Recommendations	The normalization code must be removed anyway as the decimals are correct before the execution, it is unnecessary, thus no fix is needed for this specific issue.
Comments / Resolution	Fixed - Normalization logic has been removed, price is returned as it is queried from the VolatilityOracle

Issue_03	Incorrect natspec for the setTimewindow() function
Severity	Informational
Description	Natspec says: Sets the token address for which the price is being fetched, but this is incorrect, this function updates the timeWindow, not the tokenAddress.
Recommendations	Consider updating the natspec to match the function's behavior.
Comments / Resolution	Fixed by following recommendations.

Issue_04	Configuring an Oracle with not enough observations for the specified timeWindow can cause execution to revert
Severity	Informational
Description	If the configured Oracle does not have enough observations compared to the timeWindow, the query to the Oracle will revert.
Recommendations	Consider ensuring the oracle has enough observations to not revert because of the timeWindow.
Comments / Resolution	Acknowledged.

Issue_05	Algebra Oracles can revert in case the TWAP timepoints starts to override values on the circular buffer oracle and timeWindow on PriceFeed is not adjusted.
Severity	**Informational**
Description	Algebra stores all "TWAP timepoints" into a circular buffer oracle More specifically, this is whenever a swap in a new block is triggered. The property of the circular buffer is that once the threshold of 65536 is reached, it will start writing at the very first index again. It essentially overrides that value. This is where the problem arises. To fetch the TWAP entry for a specific time can result in problems if that time is too far in the past (ie. if the desired timestamp is older than the oldest entry in the circular buffer) It will then revert here:
Recommendations	Constantly monitor the PriceFeed to ensure it does not fall into this DoS problem. In case the Algebra Oracle starts to override values on its circular buffer, make sure to update the timeWindow on the PriceFeed to fix the DoS.
Comments / Resolution	Acknowledged.

OptionsToken

The OptionsToken is a contract that enables users to exercise their right to purchase an underlyingToken at a specified price, with potential discounts based on a buffer system.

Users receive minted optionTokens to represent their right to exercise the purchase of the configured underlyingToken.

OptionToken (oGrail) is the ERC20 token associated with the OptionsToken contract.

- Admin mints optionTokens to designated addresses
- OptionTokens are required to exercise the right to purchase underlying tokens.
    - During exercise, users can receive either xGrail or liquid grail.
    - Payments made when exercising the right to purchase underlyingToken are sent to the treasury.
        - Users may obtain a discount from the current market price, based on the buffer and the discounting mechanism of the OptionPricing contract.

OptionsTokens utilize the PriceFeed to determine the amount of PaymentToken needed to purchase a specified amount of UnderlyingToken to exercise.

oTokens are 1:1 with the underlyingToken being exercised.

- For each underlyingToken a user wishes to acquire, they need one optionToken.

Users exercising optionTokens can receive either grail or xGrail.

- If users decide to receive grail, they have to pay a fee for instantly getting access to liquid grail
- If users decide to receive xGrail, they don't need to pay anything

Core Invariants:

INV 1: oTokens are 1:1 with the underlyingToken

INV 2: Users pay a fee when receiving liquid Grail

INV 3: Users don't pay a fee when receiving xGrail

INV 4: Users can get a discount based on the OptionsPricing buffer system

INV 4: Users require oTokens to exercise their right to acquire underlyingToken

Privileged Functions

- mint
- setOracle
- setTokenAdmin
- setTreasury

Issue_06	Options can be exercised for free
Severity	**Low**
Description	In the _exercise() function the paymentAmount rounds down allowing users to pay less for exercising their options than they should. With small amounts this can round down to zero.
Recommendations	Consider rounding up when calculating the paymentAmount.
Comments / Resolution	Fixed - paymentAmount is now rounded up.

OptionsPricing

The OptionsPricing contract handles the pricing of OptionTokens (oTokens) using a buffer system that tracks the volume of exercised options since the timestamp of the last purchase.

The discount should incentivize the exercise of optionTokens by providing a higher discount when the buffer is low.

The buffer is replenished gradually over time through a decay function that decreases the buffer from its current level toward zero as time passes since the last purchase.

Appendix: calculateDiscount

The calculateDiscount is the discount to apply based on a given buffer value

1. If maxCapacity or buffer value equals 0, applies maximum discount
2. If buffer value exceeds maxCapacity, applies minimum discount
3. Discount for buffer values lower than maxCapacity should be inversely proportional to the capacity's percentage

Appendix: getCurrentBufferDecay

The decay applied to the current buffer is calculated by multiplying the timeDelta from the lastPurchaseTimestamp to the current timestamp by the bufferDecayPerSecond.

- If decayAmouny exceeds the currentBuffer, then, the buffer post decay is returned 0
- if decayAmount is lower than the currentBuffer, then, the buffer post decay is returned as `currentBuffer - decayAmount`

Core Invariants:

INV 1: If buffer exceeds maxCapacity, applied discount is the minimum

INV 2: If there is no maxCapacity or buffer is 0, applied discount is the maximum

INV 3: If buffer has not reached maxCapacity, applied discount should be inversely proportional to the capacity percentage

INV 4: Buffer is replenished as time passes since the last purchase

Privileged Functions

- setMinDiscountInBps
- setMaxDiscountInBps
- setBufferDecayPerSecond
- setWorker

Issue_07	Discount calculation is wrong as it computes the discount directly proportional to the buffer
Severity	**High**
Description	calculateDiscount() calculates incorrectly the discount to apply for buffer values below the maxCapacity because the computed discount is directly proportional to the capacity's percentage, when it should be inversely proportional, correct calculation should be: - the lower the buffer, the higher the discount - the higher the buffer, the lower the discount But, the current discount calculation for capacity below the maxCapacity is the opposite. The following code calculates the discount: *function calculateDiscount(uint256 buffer) public view returns (uint256) {* *uint256 capacityCap = maxCapacity;* *if (capacityCap == 0) return maxDiscountInBps;* *if (buffer == 0) return maxDiscountInBps;* *uint256 effectiveBufferForDiscount = Math.min(buffer, capacityCap);* *if (effectiveBufferForDiscount >= capacityCap) return minDiscountInBps;* *uint256 discount = (maxDiscountInBps * effectiveBufferForDiscount) / capacityCap;* *return discount;* *}* The code correctly handles the bound - if buffer is 0, we return the maximum discount and if buffer is >= the capacity cap, we turn the minimum discount. However, the range between 0 and the cap

	causes a completely incorrect result. Instead of the discount going up with the buffer going down, which is the intention of the code, the discount goes down with the buffer going down, even reaching a discount lower than the minimum one. Then, it randomly jumps to the max discount when the buffer reaches 0.
Recommendations	Consider updating the formula to calculate the discount to calculate the discount inversely proportional to the current buffer.
Comments / Resolution	Fixed - New formula computes the correct discount based on the provided buffer.

Issue_08	Users can pay significantly less payment tokens by splitting their exercises into multiple smaller ones
Severity	**Medium**
Description	The option pricing mechanism implements the following code: *currentBuffer = bufferAfterDecay + exerciseAmount;* *lastPurchaseTimestamp = block.timestamp;* *discountInBps = calculateDiscount(currentBuffer);* The lower the currentBuffer, the bigger the discount. However, the code makes it so users that split their exercises into multiple smaller amounts will get a much better discount and will thus pay less payment tokens, compared to users to exercise their amount all at once. If we imagine that the buffer is currently at 0 and a user exercises a large amount, sufficient to go to the max capacity, he would receive the minimum discount, applied to his full amount. However, if the user splits his amount into 10 exercises, only the last exercise will have the minimum discount applied to it while all of the other ones will receive a bigger discount (e.g. the first exercise will receive a discount very close to the maximum one).
Recommendations	This issue was pre-audit acknowledged by the Camelot team and will not be fixed as it requires refactoring the code.
Comments / Resolution	Acknowledged.

Issue_09	Using getCurrentDiscountInBps() might be misleading
Severity	**Low**
Description	The function getCurrentDiscountInBps() returns the current discount without applying new buffer (buffer after time decay and exercise amount) , therefore the returned value from getCurrentDiscountInBps() won't be the exact discount applied when exercising since the actual discount takes the new buffer into consideration.
Recommendations	Consider adding an amount parameter too in the current functionality to depict a more accurate discount.
Comments / Resolution	Fixed.

Issue_10	Users can never benefit from the maximum discount
Severity	**Low**
Description	For users to benefit from the maximum discount, the buffer must be 0 (input in calculateDiscount()): *if (buffer == 0) return maxDiscountInBps;* However, the following code in *recordExerciseAndGetDiscount()* makes that impossible: *currentBuffer = bufferAfterDecay + exerciseAmount;* *lastPurchaseTimestamp = block.timestamp;* *discountInBps = calculateDiscount(currentBuffer);* Even if bufferAfterDecay is 0 (i.e. the buffer has decayed to 0), exerciseAmount will always make currentBuffer larger than 0, thus the maximum discount is unusable.
Recommendations	Since we are of the opinion that this can be considered as a design-choice, we recommend acknowledging it.
Comments / Resolution	Fixed by slightly refactoring the function and including the current exercise amount into the buffer decay calculation.

Issue_11	Incorrect discounts are applied after setBufferDecayPerSecond is called.
Severity	**Low**
Description	When the setBufferDecayPerSecond function is called it will modify the bufferDecayPerSecond without accounting for the pending decay. Resulting in a step wise change in the currentBuffer. This can lead to an unexpectedly worse or better discount. Neither of which is good as it is not representative of the true discount.
Recommendations	When calling the setBufferDecayPerSecond function, update the currentBuffer based on the pending decay before setting the new bufferDecayPerSecond.
Comments / Resolution	Fixed by following recommendations.

Issue_12	Unnecessary Math.min() usage
Severity	**Informational**
Description	The following code can be seen in calculateDiscount(): *uint256 effectiveBufferForDiscount = Math.min(buffer, capacityCap);* *if (effectiveBufferForDiscount >= capacityCap) return minDiscountInBps;* *uint256 discount = (maxDiscountInBps * effectiveBufferForDiscount) / capacityCap;* The math.min() usage is redundant as the if check below it handles a buffer over the cap anyway by returning early.
Recommendations	A solution can be to remove the Math.min() invocation as well as the effectiveBufferForDiscount variable. However, since this is non-critical, we recommend acknowledging this issue to not introduce code-changes for the resolution round on this part.
Comments / Resolution	Fixed - calculateDiscount() was refactored and now does not uses Math.min().

Issue_13	lastPurchaseTimestamp is incorrect on the first exercise.
Severity	**Informational**
Description	The OptionPricing contract does not initialize the lastPurchaseTimestamp, leading to an incorrect timeDelta upon the first exercise. This incorrect value will be resolved once the exercise is complete, and the incorrect value will not impact the following calculations since the currentBuffer will be 0.
Recommendations	When initializing the OptionPricing contract, consider setting the lastPurchaseTimestamp to block.timestamp.
Comments / Resolution	Acknowledged.

Issue_14	decay is applied immediately to new exercised amount
Severity	Informational
Description	The decay is now applied immediately to the amount being exercised, in case the decay is bigger than currentBuffer + exerciseAmount, the buyer will get the maximum discount because the decay was applied retroactively to the user's exercise amount
Recommendations	Consider acknowledging this issue if this is the intended behavior, otherwise, adjust the logic to not apply retroactively the decay to the amount being exercised
Comments / Resolution	Acknowledged..

MainToken

The MainToken contract is an upgradeable ERC20Burnable token.

It is considered as a liquid version of EscrowToken.

- Convertible to EscrowToken via the EscrowToken contract, with the ability to redeem EscrowToken back to MainToken.

The initialSupply is minted to the owner.

Core Invariants:

INV 1: Callers can burn their own tokens via the burn()

INV 2: Callers can burn other account's tokens via the burnFrom() as long as they have allowance

No issues found.

EscrowToken

The EscrowToken contract represents the illiquid version of the MainToken.

xGrail can only be obtained by converting Grail.

- When Grail is converted to xGrail, the converted amount of Grail tokens is locked within the EscrowToken contract.

xGrail can then be redeemed for Grail after a vesting period, unless the user is a whitelisted redeemer; in that case, the whitelisted redeemer can redeem xGrail for Grail instantly.

- When the redemption period concludes, users can finalize the redemption, resulting in the burning of xGrail and the receipt of Grail tokens.
- Users also have the option to cancel a redemption; in this case, they will receive their xGrail back, and the redeemEntry will be removed from the system.

Users can allocate and deallocate xGrail tokens to external contracts. Any external contract is out of scope, we highly recommend extending the audit scope to achieve full coverage.

- During allocation, the user's xGrail tokens are transferred to the EscrowToken contract.
- During deallocation, the user receives their xGrail tokens minus any applicable deallocationFee.

Core Invariants:

INV 1: It is non-transferable, except from/to whitelist addresses

INV 2: 1:1 to MainToken

INV 3: xGrail can be converted 1:1 to GRAIL through a vesting process

INV 4: Whitelisted redeemers can instantly convert xGrail to GRAIL

INV 5: Holders can allow plugins (usageContracts) to allocate their xGrail on their behalf

INV 6: If a xGrail token is burnt, so it is a GRAIL token.

Privileged Functions

- updateTransferWhitelist
- updateRedeemWhitelist
- updateDeallocationFee
- setWorker

Issue_15	usageAddress can sandwich a call to approveUsage to grief the user
Severity	**Medium**
Description	Consider the following case → 1.) UserA has given usageContractA 100 amount approval. 2.) UserA wants to decrease the approval to 50 and calls approveUsage() with an amount as 50. 3.) usageContractA frontruns the above call and firstly allocates (using allocateFromUsage()) 100 which reduces usageApprovals[][] to 0 , then the userA tx executes and usageApproval is now assigned to 50. 4.) usageContractA can again allocate this extra 50 , all in all usagecontract allocated 150 leveraging this. In addition to usageContractA leveraging this to have more allocation from userA it would also result in userA paying more fees when deallocating since deallocation fees is directly dependent on the allocation amount.
Recommendations	Consider adding a bool flag in the approveUsage function , if set to true it adds to the current approval , if false it subtracts the amount from the current approval amount .
Comments / Resolution	Fixed - A new bool variable has been added to allow users specify if they want to add or subtract approval.

Issue_16	Malicious usageAddress can grief deallocation by reverting on the deallocate function
Severity	**Low**
Description	To deallocate xGRAIL that was allocated to an usageAddress is required to do an external call to IEscrowTokenUsage(usageAddress).deallocate(), but, a malicious usageAddress can explicitly revert these calls and cause a deallocation to not be possible.
Recommendations	Consider executing the call to usageAddress in a try-catch.
Comments / Resolution	Acknowledged.

Issue_17	A race condition can cause users to receive less xGRAIL than expected
Severity	**Low**
Description	Upon deallocating, users can get charged a fee: *uint256 deallocationFeeAmount = (amount * $.usagesDeallocationFee[usageAddress]) / 10000;* That fee can be changed at any moment by an owner: *function updateDeallocationFee(address usageAddress, uint256 fee) external onlyOwner {* *if (fee > MAX_DEALLOCATION_FEE) revert DeallocationFeeTooHigh();* *_getEscrowTokenStorage().usagesDeallocationFee[usageAddress] = fee;* *emit UpdateDeallocationFee(usageAddress, fee);* *}* This allows the following to happen: 1. User deallocates, expecting a 10% fee 2. At the same time, an owner changes the fee to 15% (not in a malicious manner, he would have changed it anyway)

	3. A race condition situation is happening and the owner's transaction executes first 4. User is charged a 15% fee and receives less than expected
Recommendations	Consider communicating any change with the community upfront.
Comments / Resolution	Acknowledged.

Issue_18	Fee portion an be redirected to user via rounding logic
Severity	**Low**
Description	The deallocationFeeAmount rounds down, leading to users paying less than the actual deallocation fee. In cases with very small amounts being deallocated the fee can be avoided completely.
Recommendations	Consider rounding up when calculating the deallocation fee. Consider making sure this does not introduce arithmetic reverts in future subtractions.
Comments / Resolution	Fixed by following recommendations.

Issue_19	EscrowToken.burn is unusable due to _update() override
Severity	Informational

Description	EscrowToken is non-transferable token but seems to allow users to burn their XGrail and corresponding Grail tokens via the burn() function
	*/**
	* @notice Burns a specific amount of tokens from caller's account
	* @param amount The amount of token to be burned
	*/
	function burn(uint256 amount) public virtual override(ERC20BurnableUpgradeable, IEscrowToken) {
	_getEscrowTokenStorage().MainToken.burn(amount);
	ERC20BurnableUpgradeable.burn(amount);
	}
	The issue is that _update() function doesn't account for burning

Recommendations	Consider adding to != address(0) to the check
Comments / Resolution	Fixed by following recommendations.

Issue_20	Old approvals can be left hanging
Severity	**Informational**
Description	Consider the following case → a.) UserA converts his GRAIL to XGRAIL. b.) UserA approves usageContractA for his XGRAIL using approveUsage(). c.) UserA decides to not make any allocation , redeems the XGRAIL back to GRAIL with redeem(). d.) usageApprovals[][] still stores the old approval to the usageContractA , in case where userA converts his GRAIL to XGRAIL in future the usageContractA can still make an allocation for userA which would be unintended since userA does not trust usageContractA anymore.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_21	Incorrect value for amountReturnedToUser parameter when emitting Deallocate event.
Severity	**Informational**
Description	The amount returned to the user in a deallocation is the amount allocated - deallocationFeeAmount , but the value assigned to amountReturnedToUser in the Deallocate event does not account for the deallocationFees.
Recommendations	Assign amount - deallocationFeeAmount as the amountReturnedToUser parameter. It must be ensured that there can never be any underflow when following this recommendation, as it would essentially result in an operation revert via event.
Comments / Resolution	Acknowledged.

Issue_22	No function signature differentiation can cause serious issues
Severity	**Informational**
Description	The impact of this issue is currently non-existent, however even a slight change in the code could result in this being exploitable. The following function can be seen in EscrowToken: *function allocate(address usageAddress, uint256 amount, bytes calldata usageData) external {* *_allocate(msg.sender, usageAddress, amount);* *// allocates xGRAIL to usageContract* *IEscrowTokenUsage(usageAddress).allocate(msg.sender, amount, usageData);* } Important logic is conducted in the internal _allocate() function which should not be manipulated. However, the issue is that the call to the usage contract has the exact same function signature that the EscrowToken::allocate() has (both are allocate(address,uint256,bytes). This allows a malicious user to set the usage contract as the EscrowToken contract itself, resulting in the msg.sender in that call to be EscrowToken while the usageAddress would be the malicious user, effectively causing the EscrowToken to allocate to the malicious user.
Recommendations	Consider acknowledging this issue, and making sure that the approval mechanism remains in place to prevent critical issues.
Comments / Resolution	Acknowledged.

Issue_23	Incorrect revert error in EscrowToken::_allocate()
Severity	**Informational**
Description	Upon allocating in EscrowToken, the following code can be seen: *if ($.usageApprovals[userAddress][usageAddress] < amount) revert InsufficientAllocatedAmount();* If the approval is insufficient, we revert with InsufficientAllocatedAmount(). However, this is an incorrect error as we revert due to insufficient approval, not insufficient allocation.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Fixed by following recommendations.

Issue_24	Insufficient output when getting a users token balance
Severity	**Informational**
Description	The getXGrailBalance function is intended to return a user's balance of XGrail. But it ignores the actual user balance of XGrail. Instead it only returns the balance.allocatedAmountand the balance.redeemingAmount.
Recommendations	Consider elaborating on this issue.
Comments / Resolution	Acknowledged.

Issue_25	EscrowToken is Missing Zero Address Check
Severity	**Informational**
Description	`EscrowToken` upon initialization does not validate that `MainToken` is not address(0). Allowing bad or incorrect configurations to pass through the initialization process.
Recommendations	Validate that `EscrowToken` is not address(0).
Comments / Resolution	Fixed by following recommendations.

Issue_26	ApproveUsage event does not accurately logs if approval was added or subtracted.
Severity	**Informational**
Description	The usage approval can now be either added or subtracted, but, the ApproveUsage event does not includes the add variable to inform if the approval was added or subtracted.
Recommendations	Consider including the add variable on the ApproveUsage event.
Comments / Resolution	Fixed by following recommendations.

TokenSale

The TokenSale contract is a contract that facilitates the sale of tokens in exchange for ETH or WETH.

- The collected WETH and ETH are directed to the treasury.

A whitelist utilizing MerkleProofs can be established to identify users eligible for a discount.

Purchased tokens are distributed according to a ratio between unlocked and locked tokens.

The tokens available for sale must be funded after deploying the TokenSale contract to enable purchases.

Admins have the ability to recover tokens from the TokenSale contract.

Appendix: calculateRatio

The ratio is applied to the amount of tokens being purchased. It determines the amount of lockedTokens and unlocked tokens to distribute.

➔ amount - amount being purchases
➔ ratio - ratio of locked tokens distributed in basis points (1/100th of a percent)
➔ BP - Basis Points (1/100th of a percent)

Core Invariants:

INV 1: purchases paid with ETH or WETH

INV 2: Whitelisted users get a discount

INV 3: Admins can recover funds from the TokenSale

INV 4: unlocked and locked tokens are distributed based on the specified ratio

Privileged Functions

- recoverTokens

Issue_27	Not refunding excess native when purchasing tokens with ETH
Severity	**Low**
Description	When purchasing tokens via buyTokens() with ETH, the msg.value should be greater than the totalCost calculated , but the excess ETH is not returned to the user, instead, it is sent to the treasury. Users might overpay in scenarios where they are unsure of the price after the discount.
Recommendations	Consider refunding the excess ETH sent by the user when purchasing tokens with ETH.
Comments / Resolution	Fixed by following recommendations.

Issue_28	totalCost is calculated with the decimals of the purchaseToken instead of decimals of ETH
Severity	**Low**
Description	For example, a purchaseToken w/ 6 decimals that costs 1 ETH getAmountCost() will do (without discount): 1e18 * 1e6 / 1e18 => 1e6 - cost for 1 purchaseToken would be calculated as 1e6 instead of 1e18 ETH.
Recommendations	Divide the costBeforeDiscount by the decimals of the token being sold instead of diving by WAD
Comments / Resolution	Acknowledged.

Issue_29	locked and unlocked tokens can get stuck on the TokenSale contract
Severity	**Low**
Description	When recovering locked or unlocked tokens, the amount that can be pulled out of the contract is determined by the saleAmount and the ratio, but, this can be problematic because it is possible that this calculation won't allow to recover all the locked and unlocked tokens.
Recommendations	Consider defining a deadline for the tokenSale, and, instead of calculating the ratio for locked and unlocked tokens when recovering them, best to allow recovering all the balance that was left after the tokenSale has ended.
Comments / Resolution	Acknowledged.

Issue_30	Buyers can purchase tokens at zero cost
Severity	**Low**
Description	The getAmountCost function rounds down when determining the cost, which can allow an attacker to buy tokens for less than they should be able to and in the worst case for a cost of 0.
Recommendations	When calculating getAmountCost consider rounding up.
Comments / Resolution	Fixed by following recommendations.

Issue_31	Ratio favors users as it rounds towards unlocked tokens
Severity	Low
Description	Because the _calculateRatio rounds down a greater than expected ratio of unlocked tokens will be distributed.
Recommendations	If receiving unlocked tokens favors the user then consider rounding up so that users are not at an advantage and are required to take on more locked tokens.
Comments / Resolution	Fixed by following recommendations.

Issue_32	Merkle Root should be changeable in cases where a user needs to be removed from the whitelist
Severity	Informational
Description	The merkle root is currently immutable and once set in the constructor can not be changed . Therefore it is impossible to edit the merkle root if a user is deemed to be blacklisted or not eligible afterwards or new users are supposed to be added to the whitelist.
Recommendations	Consider adding a setter for the merkle root where the owner can update to the new merkle root computed offchain. Potential impact of changing root must be considered.
Comments / Resolution	Acknowledged.

Issue_33	wethRaised is not updated for purchases paid with ETH
Severity	Informational
Description	WETH == ETH, but, purchases paid with ETH are not accounted for on the wethRaised variable. This means that wethRaised won't accurately track the raised amount during the token sale.
Recommendations	Consider updating wethRaised with the amount of ETH paid for the purchase of tokens.
Comments / Resolution	Acknowledged.