

REVEAL<sup>3</sup>

# AmmoCrypt

Security Audit

## 1. Disclaimer

Security audits cannot uncover all existing vulnerabilities; even an audit in which no vulnerabilities are found is not a guarantee of a secure system.

However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it.

Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code audits aimed at determining all locations that need to be fixed. Within the customer-determined time frame, we performed an audit in order to discover as many vulnerabilities as possible.

The focus of our audit was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself.

Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

## **2. Terminology & Risk Classification**

For the purpose of this audit, we adopt the following terminology: To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

**Likelihood:** The likelihood of a finding to be triggered or exploited in practice.

**Impact:** The technical and business-related consequences of a finding.

**Severity:** An additive consideration based on the Likelihood and the Impact (as referenced below).

We use the table below to determine the severity of any and all findings. Please note that none of these risk classifications constitute endorsement or opposition to a particular project or contract.

	<b><u>Impact</u></b>		
<b><u>Likelihood</u></b>	High	Medium	Low
High	Severity: Critical	Severity: High	Severity: Medium
Medium	Severity: High	Severity: Medium	Severity: Low
Low	Severity: Medium	Severity: Low	Severity: Low

### **3. Summary**

#### **Contracts In-scope:**

- **YEET.sol**
- **Vesting.sol**
- **TransferController.sol**
- **PrimarySales.sol**
- **BaseToken.sol**
- **IVesting.sol**
- **ITransferController.sol**
- **ITokenFactory.sol**
- **IPrimarySales.sol**
- **IERC20Burnable.sol**
- **IBaseToken.sol**

In the period 6 Nov. 2023 - 20 Dec. 2023 we audited AmmoCrypt's smart contract.

In this period of time a total of 23 of issues were found.

<b>Severity</b>	<b>Issues</b>
Critical Risk	0
High Risk	0
Medium Risk	4
Low Risk	6
Gas Optimization	4
Informational	9

## 4. Issues

### **[Informational - 001]: Pragma Version Incompatibility**

Description: The smart contract currently employs pragma version 0.8.18. However, this version is incompatible with the libraries being used, which require version 0.8.20. This version discrepancy can lead to potential compatibility issues and unexpected behavior in the contract's execution.

Remediation: To ensure compatibility and optimal functionality of the smart contract, it is recommended to either update the pragma directive to specifically use version 0.8.20 or to employ the caret (^) notation, ^0.8.20.

Affected contracts: All ( - Vesting)

Status: *The issue was mitigated.*

### **[Informational - 002]: Incorrect Function Usage for Role Management in OpenZeppelin's AccessControl.sol**

Description: The smart contract incorrectly utilizes the `_setupRole` function for role assignment within the context of OpenZeppelin's `AccessControl.sol`. This function is not a valid method for granting roles in the `AccessControl` implementation. The misuse of this function could lead to improper role management, potentially compromising the access control mechanisms of the contract.

Remediation: To align with the intended usage of OpenZeppelin's `AccessControl.sol`, it is advised to replace the `_setupRole` function with the appropriate methods provided by the library. Specifically, use `_grantRole(bytes32 role, address account)` for granting access roles, and `_revokeRole(bytes32 role, address account)` for revoking them.

Affected contracts: All ( - Vesting)

Status: *The issue was mitigated.*



```
1  _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
2  _setupRole(MultisigHelper.MINTER_ROLE, minter);
3  _setupRole(MultisigHelper.DEPLOYER_ROLE, deployer);
4  _setupRole(MultisigHelper.YEET_BURNER_ROLE, deployer);
```

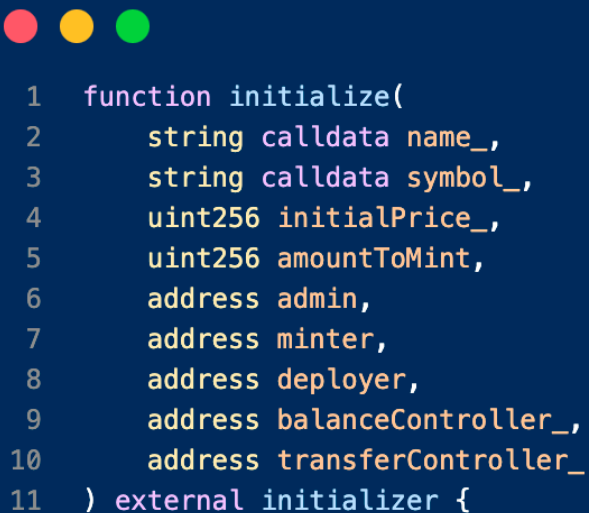
## [Informational - 003]: Stack Too Deep in the 'initialize' Function

Description: The 'initialize' function in the smart contract encounters a 'Stack Too Deep' error. This issue arises due to the excessive number of local variables and parameters, which exceeds the EVM's stack limit within a single function context. The current implementation with multiple parameters passed as calldata contributes to this limitation breach.

Remediation: To mitigate this issue, consider changing the data location of string parameters from calldata to memory. Utilizing memory for complex data types like strings can help reduce the stack depth, as memory variables do not contribute to the EVM's stack limit in the same way calldata variables do. This change should help to avoid the 'Stack Too Deep' error by optimizing the function's stack usage.

Affected contracts: YEET.sol, BaseToken.sol

Status: The issue was mitigated.



```
1  function initialize(  
2      string calldata name_,  
3      string calldata symbol_,  
4      uint256 initialPrice_,  
5      uint256 amountToMint,  
6      address admin,  
7      address minter,  
8      address deployer,  
9      address balanceController_,  
10     address transferController_  
11 ) external initializer {
```

## **[Informational - 004]: Unnecessary Use of OwnableUpgradeable**

Description: The smart contract incorporates the OwnableUpgradeable contract from OpenZeppelin and initializes it, but it does not utilize the onlyOwner modifier in any of its functions. This inclusion of OwnableUpgradeable without leveraging its primary feature – access control via the onlyOwner modifier – results in redundant code and unnecessary contract complexity. Such unused components can increase the contract size, leading to higher deployment and execution costs, and potentially introduce unmaintained code paths that could harbor unnoticed vulnerabilities.

Affected contracts: All

*Status: The issue was mitigated.*

## **[Informational - 005]: Verification of Addresses Against Dead or Zero Address**

Description: The current implementation of the smart contract does not validate whether the addresses provided in certain functions (such as the constructor, addToBlacklist, or removeFromBlacklist) are the dead address (0xdead) or the zero address (0x0). Utilizing these addresses can lead to unintended behaviors, security vulnerabilities, or could render certain functionalities inoperative. Ensuring that input addresses are neither the zero address nor the dead address is a crucial aspect of smart contract security and functionality.

Affected contracts: YEET.sol

*Status: The issue was mitigated.*

## [Low - 001]: Lack of Interface Implementation Verification for transferController

Description: The smart contract currently does not verify whether the transferController address provided implements the ITransferController interface. This oversight could lead to potential issues if an incorrect contract is linked, as the expected functions and behaviors defined in the ITransferController interface might not be present. Ensuring that any contract linked as a transferController adheres to the ITransferController interface is crucial for the contract's integrity and functionality.

Remediation: Modify the constructor to include a check that ensures transferController supports the ITransferController interface. This can be done using the supportsInterface function provided by IERC165.

```
1 bool _sI = IERC165(transferController_).supportsInterface(type(ITransferController).interfaceId);  
2 require(_sI, 'Invalid contract interface');
```

Affected contracts: YEET.sol + BaseToken.sol

Status: The issue was mitigated.

## [Informational - 006]: Uninitialized ERC165Upgradeable in the Constructor

Description: The smart contract includes an import of ERC165Upgradeable from OpenZeppelin's upgradeable contracts collection, but fails to initialize this contract in the constructor. ERC165Upgradeable is a critical component for interface compliance checking (as per EIP-165), and its initialization is essential for the proper functioning of these checks. The absence of its initialization can lead to incorrect behavior or failures in interface compliance verification, potentially affecting the contract's interaction with other contracts and interfaces.

Remediation: There are two possible courses of action, depending on the intended use of ERC165Upgradeable in the smart contract

**If ERC165Upgradeable is not intended for use:** Remove the import statement of ERC165Upgradeable from the contract. This step will clean up the contract code, removing unnecessary dependencies and reducing potential confusion or misinterpretation regarding the contract's functionality.



**If ERC165Upgradeable is required for the contract's functionality:** Properly initialize ERC165Upgradeable in the constructor. This can be achieved by adding the initialization function `__ERC165_init()` within the constructor. This step ensures that all the setup and configuration required for ERC165Upgradeable are correctly implemented, enabling the contract to utilize EIP-165 interface compliance checks effectively.

Affected contracts: YEET.sol, TransferController.sol, TokenFactory.sol, PrimarySales.sol

*Status: The issue was mitigated.*

### **[Informational - 007]: Lack of Enforcement Mechanism for Whitepaper Stipulations in the Smart Contract**

Description: The project's whitepaper stipulates that initial YEET tokens are solely for Asset Token mining and cannot be traded or sold. However, the current smart contract lacks mechanisms to enforce this, allowing these tokens to be potentially sold or traded contrary to the whitepaper's intentions. Moreover, there's a risk associated with the blacklisting process; users might transfer tokens before being blacklisted, defeating its purpose.

Affected contracts: YEET.sol

*Status: The issue is not valid. The protocol removed that functionality from the smart contract.*

### **[Informational - 008]: Incorrect Import Statement for OpenZeppelin's ERC165Upgradeable**

Description: The current import statement used in the smart contract for OpenZeppelin's ERC165Upgradeable is invalid. The statement attempts to import both `IERC165Upgradeable` and `ERC165Upgradeable` from the same file, which is not the correct format for importing these components.

Affected contracts: TransferController.sol

*Status: The issue was mitigated.*

## **[Low - 002]: Verification of Address Validity and Role Assignment**

Description: The smart contract currently does not include checks to verify if inputted addresses in certain functions (like the constructor, addToBlacklist, removeFromBlacklist) are neither the dead address (0xdead) nor addresses with significant roles within the smart contract. Ensuring that these addresses are valid and do not hold crucial roles is essential for maintaining the contract's integrity and security.

Affected contracts: TransferController.sol

*Status: The issue was mitigated.*

## **[Gas - 001]: Potential Gas Limit Exceeded Due to Large 'availableTokens' Array**

Description: In the smart contract, particularly within the 'PrimarySales' module, there is a concern regarding the scalability of the 'availableTokens' array. If an excessive number of clones are deployed, this array could grow significantly large, posing a risk that the 'isTokenAvailable' function may fail. The failure would be due to the gas required for execution exceeding the block's gas limit, a constraint imposed by the Ethereum network.

Remediation: Replace the array with a mapping structure. This can significantly optimize gas consumption, as mappings are more gas-efficient for lookup operations compared to large arrays.

Affected contracts: TokenFactory.sol

*Status: The issue was mitigated.*

## **[Low - 003]: Lack of Interface Implementation Check for tokenFactory**

Description: The smart contract currently does not validate if the tokenFactory address implements the ITokenFactory interface. This oversight can lead to potential integration issues if an incorrect contract is linked, as the expected functions and behaviors defined in the ITokenFactory interface might not be present. Ensuring that any contract linked as tokenFactory adheres to the ITokenFactory interface is crucial for the contract's functionality and integrity.

Remediation: Ensure tokenFactory supports the ITokenFactory interface. This can be achieved using the supportsInterface function provided by IERC165.

Affected contracts: PrimarySales.sol

*Status: The issue was mitigated.*

## **[Medium - 001]: Decimal Mismatch in Price Calculation of PrimarySales.sol**

Description: The 'processSale' function in PrimarySales.sol has vulnerabilities due to the way price is set in YEET.sol, especially when using USDC and USDT tokens on the Polygon chain, which have 6 decimals instead of 18. The formula used:

```
uint256 price = (IBaseToken(token).getPrice() * amount) / 10 ** 18;
```

leads to decimal mismatches in different cases, causing incorrect pricing during transactions.

Remediation:

For a 6-decimal price in YEET and a non-decimal amount, modify the price calculation to `uint256 price = IBaseToken(token).getPrice() * amount` and introduce a maximum price limit. Adjust token transfer to `IERC20(token).safeTransfer(msg.sender, amount * 10 ** IERC20(token).decimals());`.

For a 6-decimal price and an 18-decimal amount, retain the current function but add `require(amount > 1 ether, "Amount must be bigger!");`.

Affected contracts: PrimarySales.sol

*Status: The issue was mitigated.*

## [Medium - 002]: Lack of Address Validation in Withdraw Function

Description: The smart contract's withdraw function currently does not include a validation check to ensure that the 'to' address is neither the zero address (0x0) nor the dead address (0xdead). Without this validation, there's a risk of funds being irretrievably sent to these addresses, potentially leading to loss of assets.

Remediation:

Modify the withdraw function to include a validation check for the 'to' address. This check should confirm that the address is not the zero address or the dead address.

Implementing this safeguard will prevent accidental or malicious transactions to invalid addresses, enhancing the security and reliability of the contract.

```
1  function withdraw(  
2      address token,  
3      address to,  
4      uint256 amount  
5  ) external onlyRole(MultisigHelper.BALANCE_CONTROLLER_ROLE) {  
6      require(token != address(0), "Invalid token address");  
7      require(amount != 0, "Amount must be greater than zero");  
8      require(  
9          address(acceptedTokens[token]) != address(0) || isTokenAvailable(token) || token == yeet,  
10         "Incorrect token"  
11     );  
12  
13     // @audit Useless code because the transaction will automatically revert if the contract doesn't have enough tokens  
14     require(ERC20(token).balanceOf(address(this)) >= amount, "Contract token balance must be above the amount");  
15  
16     isTokenAvailable(token) || token == yeet  
17         ? ERC20(token).safeTransfer(to, amount)  
18         : acceptedTokens[token].safeTransfer(to, amount);  
19  
20     emit Withdrawal(token, to, amount);  
21 }
```

Affected contracts: PrimarySales.sol

Status: *The issue was mitigated.*

## **[Gas - 002]: Gas Limit Exceeded in 'isTokenAvailable' Function**

Description: The 'isTokenAvailable' function in the smart contract poses a risk of exceeding the block gas limit if the array of tokens becomes excessively large. This issue can cause functions that call 'isTokenAvailable' to fail, particularly affecting the 'withdraw' function, which calls it twice. This problem arises from the function's design, which requires more gas as the token array grows, potentially reaching a point where it exceeds the maximum gas limit of a block.

Affected contracts: PrimarySales.sol

*Status: The issue was mitigated.*

## **[Informational - 009]: Missing Return Statements in Function Handling 'tokens' Array**

Description: The current implementation of the function handling the 'tokens' array in the smart contract lacks two critical return statements. Firstly, it does not return false when the 'tokens' array length is zero. Secondly, there is no return false statement after the completion of the for loop, which is a recommended best practice for clarity and explicitness in contract behavior.

Affected contracts: PrimarySales.sol

*Status: The issue was mitigated.*

## **[Low - 004]: Risk of Overflow in Casting from uint256 to uint128 and uint64**

Description: The smart contract contains instances where uint256 values are cast to uint128 and uint64 types. This presents a risk of overflow when the uint256 values exceed the maximum limits of uint128 and uint64. Such overflows can lead to unintended contract behavior and vulnerabilities.

This issue is present in both the 'setYeetRate' and 'setBurnPercentage' functions.

Remediation: Utilize OpenZeppelin's SafeCast library to safely cast from uint256 to smaller integer types. SafeCast provides functions that perform checks before casting, ensuring that the value does not exceed the target type's range and preventing overflows. Replace the direct casts with the appropriate SafeCast functions in both 'setYeetRate' and 'setBurnPercentage' to enhance the safety and reliability of these operations.

Affected contracts: BaseToken.sol

*Status: The issue was mitigated.*

### **[Medium - 003]: Inadequate Balance Check Before Burning Tokens**

Description: In the current implementation, the contract checks if `yeet.balanceOf(msg.sender) >= amount * yeetRate` before burning YEET tokens. However, the actual amount burned is calculated differently:  $(\text{amount} * \text{yeetRate} * \text{burnPercentage}) / \text{BASIS\_POINTS}$ . This discrepancy can lead to a situation where the sender's YEET balance is sufficient for the initial check but insufficient for the actual burn amount, especially if the `burnPercentage` is higher than `BASIS_POINTS`.

Remediation: Modify the balance check to align with the actual burn amount. The check should be `yeet.balanceOf(msg.sender) >= (amount * yeetRate * burnPercentage) / BASIS_POINTS`. This ensures that the sender has enough YEET tokens for the burn process, preventing potential issues due to insufficient balance. By adjusting this check, the contract's logic will accurately reflect the intended token burn mechanics, enhancing its reliability and correctness.

Affected contracts: BaseToken.sol

*Status: The issue was mitigated.*

### **[Low - 005]: Potential Failure in 'mint' Function Due to Insufficient Allowance**

Description: In the smart contract, the function `mint(uint256 amount)` includes the line `yeet.burnFrom(msg.sender, amountToBurn);`. This operation might fail if `msg.sender` has not granted sufficient allowance to the `baseToken` smart contract. The `burnFrom` function requires that the `msg.sender` (whether a wallet or another smart contract) has explicitly approved the `baseToken` contract to spend tokens on its behalf. Without this allowance, the `burnFrom` call will revert, causing the entire `mint` function to fail.

Affected contracts: BaseToken.sol

*Status: The issue was mitigated.*

## **[Medium -004]: Insufficient Validation of 'startTime' in 'createRoundByWallet'**

Description: The createRoundByWallet function in the smart contract currently checks if `block.timestamp > (startTime + cliffDuration)`, but it lacks a direct validation to ensure that 'startTime' is set in the future. This can lead to scenarios where 'startTime' is set to zero (or a past time), and the condition still holds true if 'cliffDuration' is sufficiently long. Consequently, rounds could be incorrectly initiated at past timestamps, potentially causing functional discrepancies in the contract.

Remediation: Introduce an additional check in the createRoundByWallet function to explicitly verify that 'startTime' is in the future, independent of the 'cliffDuration'. This validation should ensure that 'startTime' is greater than the current block timestamp. By adding this check, the contract can prevent the creation of rounds that inadvertently start in the past.

Affected contracts: Vesting.sol

Status: -

## **[Gas - 003]: Inefficient Round ID Check Inside Loop**

Description: The current implementation checks the validity of each roundId inside a for loop using the condition `if (roundIds[i] >= roundsLength) { revert("Invalid round id"); }`. This method, while effective, can be gas-inefficient, as it involves repetitive checks for each element in the loop.

Remediation: Optimize the round ID validation by checking if `ids.length` is smaller or equal to `rounds.length` before entering the for loop.

Affected contracts: Vesting.sol

Status: -

## **[Gas - 004]: Redundant Check in the '\_claim' Function**

Description: In the smart contract, the '\_claim' function contains a check if  $(\text{block.timestamp} < \text{round.cliffDuration})$  before invoking the `_unlocked` function. However, this check is redundant as `_unlocked` performs the same validation. Additionally, the `_unlocked` function is also called in the `totalUnlocked` function, making the check in `_claim` unnecessary and inefficient.

Remediation: Remove the  $\text{block.timestamp} < \text{round.cliffDuration}$  check from the `_claim` function to eliminate redundancy and optimize gas usage. Since the `_unlocked` function already includes this check, it ensures that the necessary validation is still performed. Furthermore, in the `_claim` function, after fetching the `unlockAmount`, add a `require` statement to ensure that this amount is not zero. This will handle cases where no tokens are available for unlocking, maintaining the function's integrity.

Affected contracts: `Vesting.sol`

Status: -

## **[Low - 006]: Potential Issue with Calculating vestedAmount in Specific Scenario**

Description: In the smart contract, there's a scenario where the calculated `vestedAmount` might be zero, leading to users being unable to claim tokens. This occurs when `unlockTime` (which can be `block.timestamp` sometimes) is exactly equal to `round.cliffDuration`. In such a case, the formula

*`uint256 vestedAmount = (totalAmount * (unlockTime - round.cliffDuration)) / round.duration;`*

results in `vestedAmount` being zero because  $(\text{unlockTime} - \text{round.cliffDuration})$  equals zero.

Remediation: Modify the initial check in the function from  $\text{if } (\text{block.timestamp} < \text{round.cliffDuration})$  to  $\text{if } (\text{block.timestamp} \leq \text{round.cliffDuration})$ . This adjustment ensures that users are not able to claim tokens until `block.timestamp` is strictly greater than `round.cliffDuration`, thus preventing the scenario where `vestedAmount` calculates to zero when `block.timestamp` is exactly equal to `round.cliffDuration`.

Affected contracts: `Vesting.sol`

Status: -