Cyberscope

A TAC Security Company

Audit Report

MIYI

November 2025

Network BSC

Addresses 0x03d8ae4c2bbb9b9da7c3731e11b79be665ac8751
0x32D1ED969771aFcf3DE53E472C669185E2340D07
0x62Ea23e6C16b99B762eBD2105a7388dC1633b84A
0xB9722C7b4E23679dda98cB91b510362c7f9e97Fe

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Explorer	https://bscscan.com/address/0x03d8ae4c2bbb9b9da7c3731e11b79be665ac8751 https://bscscan.com/address/0x32D1ED969771aFcf3DE53E472C669185E2340D07 https://bscscan.com/address/0x62Ea23e6C16b99B762eBD2105a7388dC1633b84A https://bscscan.com/address/0xB9722C7b4E23679dda98cB91b510362c7f9e97Fe

Audit Updates

Initial Audit	24 Oct 2025 https://github.com/cyberscope-io/audits/blob/main/myex/v1/audit.pdf
Corrected Phase 2	05 Nov 2025

Source Files

Filename	SHA256
MIYIVesting.sol	e10c7cf9abca88910c5332a8cbf60400ef504ae9b949f081f2581b61d4527041
MIYICredit.sol	49e6baf249684b61bde4e73e64b47d3705a9892ec5d24228c546b1a518e2a41d
MIYINFTVestingPool.sol	7f463339d2cfe6fbe3ade08a583178ce011251106be6488dd8f06fb6d00f3dee
MIYINFT.sol	b394d0b2bc8d6d0ab42307124c847cb0737a6bcc1437f4ee2f701ce3c4d218d2

Overview

MIYICredit Contract

The `MIYICredit` contract implements the core functionalities of an ERC20 token with additional features for pausing and ownership management. It defines a fixed total supply of **21 billion MIYI tokens**, minted entirely at deployment and assigned to the contract owner. The token integrates transfer control and burn capabilities, ensuring both flexibility and long-term stability within the ecosystem.

Core Features

The contract leverages OpenZeppelin standards (`ERC20` , `ERC20Burnable` , `Pausable` , `Ownable`) to provide secure and auditable token operations. Token transfers can be paused and resumed by the owner to safeguard the system against potential threats or emergency conditions. Holders can burn tokens to reduce supply, while ownership privileges ensure that administrative controls remain centralized and protected.

MIYINFT Contract

The `MIYINFT` contract implements a non-fungible token (NFT) system representing unique digital assets under the MIYI ecosystem. Each token is minted according to strict supply limits. The contract allows flexible minting while maintaining long-term integrity through its locking mechanism.

Minting Functionality

The contract supports both single and batch minting, enabling efficient creation and distribution of NFTs. Only the owner can mint, ensuring that supply issuance remains fully controlled. Minting operations automatically enforce the maximum supply limit, preventing over-minting and maintaining the designed scarcity.

Mint Locking Mechanism

The contract allows the owner to permanently disable minting through the `lockMinting` function. Once locked, no new NFTs can be created, ensuring that the total NFT supply remains capped for the lifetime of the collection.

Metadata Management

The owner can update the metadata base URI via the `setBaseURI` function. The current total supply and minted count can be retrieved using the `totalMinted` function, ensuring traceability.

MIYIVesting Contract

The `MIYIVesting` contract governs the structured distribution of `MIYICredit` across multiple stakeholders through time-based vesting schedules. It ensures controlled, transparent token emission aligned with project development and ecosystem growth objectives. Each vesting schedule defines parameters such as start time, duration, and beneficiary allocation.

Initialization and Distribution

Upon initialization, the contract receives the full token supply and distributes it into multiple vesting schedules, each assigned to a specific allocation category. It creates unique schedules for ecosystem contributors, staking, team members, marketing, grants, and other strategic areas. A portion of tokens is sent directly to the NFT Vesting Pool to reward NFT holders.

Vesting Schedule Models

The contract supports several vesting models:

- **Linear Vesting:** Tokens are released gradually over the full vesting duration.
- **Platform Vesting:** Includes a short cliff followed by a two-year linear release.
- **Cliff Vesting:** Tokens are fully released after a set cliff period.
  This variety allows tailored distribution strategies across project stakeholders.

Token Release Process

Beneficiaries can claim vested tokens through the `release` function. The amount available for release is calculated based on elapsed time and the schedule's parameters. This function ensures that tokens are only released proportionally and securely through direct ERC20 transfers.

Allocation Structure

The total supply is divided into categories based on predefined percentages: Ecosystem (20%), Staking (20%), Team (13%), Platform (13%), Marketing (9%), Reserve (4%), Grants (6%), Fundraising (10%), and Cornerstone (5%, allocated to NFT Vesting Pool). This structure ensures a balanced token economy aligned with long-term project goals.

Security and Ownership

Initialization can only occur once and requires the contract to hold the total token supply. All administrative functions are restricted to the contract owner, ensuring full control over the vesting setup and preventing unauthorized modification of schedules.

MIYINFTVestingPool Contract

The `MIYINFTVestingPool` contract links NFT ownership with token vesting, allowing NFT holders to gradually claim `MIYICredit` rewards. Each NFT represents a unique allocation share from a dedicated pool of **1.05 billion MIYI tokens**, distributed linearly over a three-year period. This mechanism integrates token rewards directly into NFT ownership, promoting long-term engagement.

Allocation Management

The contract owner can assign token allocations to specific NFTs using single or batch methods. Each allocation determines how many tokens an NFT will earn over the vesting duration. Once finalized, allocations cannot be modified, ensuring a fair and immutable distribution framework.

Vesting Logic

Tokens vest linearly from the defined start time across **1,095 days (3 years)**. The amount vested for each NFT depends on how much time has passed since vesting began. When the full duration is reached, the entire allocated amount becomes claimable by the NFT holder.

Claiming Mechanism

NFT holders can claim their vested tokens once claiming is enabled by the contract owner. The `claim` function checks NFT ownership and calculates the releasable token amount. Tokens are then securely transferred to the NFT holder's address, with all claims tracked to prevent duplication.

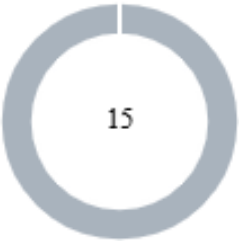
Administrative Controls

The owner maintains control over the vesting system by toggling claim availability, setting allocations, and finalizing distribution parameters. The contract enforces strict limits to prevent total allocations from exceeding the defined pool size, ensuring complete alignment between token reserves and NFT-based vesting rights.

Transparency and Monitoring

Beneficiaries can query vested, claimable, and total allocated amounts through public view functions. These tools allow real-time monitoring of vesting progress and promote transparency across all NFT holders participating in the MIYI reward system.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	15

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	0	15	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	CR	Code Repetition	Acknowledged
●	CCR	Contract Centralization Risk	Acknowledged
●	MUA	Metadata Update Authority	Acknowledged
●	MT	Mints Tokens	Acknowledged
●	MMN	Misleading Method Naming	Acknowledged
●	PSU	Potential Subtraction Underflow	Acknowledged
●	PUA	Potential Unauthorized Actions	Acknowledged
●	RISP	Redundant Individual Struct Property	Acknowledged
●	ST	Stops Transactions	Acknowledged
●	TSI	Tokens Sufficiency Insurance	Acknowledged
●	ZD	Zero Division	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L13	Divide before Multiply Operation	Acknowledged
●	L18	Multiple Pragma Directives	Acknowledged

●	L19	Stable Compiler Version	Acknowledged

CR - Code Repetition

Criticality	Minor / Informative
Location	MIYIVesting.sol#L314,365,383,401
Status	Acknowledged

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
Shell
 _createLinearSchedule(beneficiaries[0], amounts[0],
startTime, 4 * 365 days);
_createLinearSchedule(beneficiaries[1], amounts[1],
startTime, 4 * 365 days);
_createLinearSchedule(beneficiaries[2], amounts[2],
startTime, 4 * 365 days);
_createPlatformSchedule(beneficiaries[3], amounts[3],
startTime);
_createLinearSchedule(beneficiaries[4], amounts[4],
startTime, 4 * 365 days);
_createCliffSchedule(beneficiaries[5], amounts[5],
startTime, 2 * 365 days);
_createLinearSchedule(beneficiaries[6], amounts[6],
startTime, 4 * 365 days);
_createLinearSchedule(beneficiaries[7], amounts[7],
startTime, 2 * 365 days
...);
function _createLinearSchedule(address beneficiary,
uint256 amount, uint64 start, uint64 duration) internal
function _createCliffSchedule(address beneficiary, uint256
amount, uint64 start, uint64 cliffDuration) internal
```

```
function _createPlatformSchedule(address beneficiary,
uint256 amount, uint64 start) internal
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

Team Update

The team has acknowledged that this is not a security issue and states:

The repeated function calls are intentional for clarity and explicit mapping to the 9 tokenomics groups. Each call uses a distinct vesting type or duration (linear, cliff, platform), and separating them improves readability and auditability. This design choice avoids hidden logic inside loops and aligns directly with the public tokenomics breakdown. Therefore, no change is required.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	MIYICredit.sol#L891,892 MIYINFT.sol#L3644,3657,3664,3677,3682 MIYIVesting.sol#L274,291,420 MIYINFTVestingPool.sol#L426,444,460,484,491
Status	Acknowledged

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```shell
function pause() external onlyOwner
function unpause() external onlyOwner
```

```shell
constructor(string memory baseURI_, uint256 maxSupply_,
address initialOwner)
function mint(address to) external onlyOwner
function batchMint(address[] calldata toList) external
onlyOwner
function setBaseURI(string calldata newURI) external
onlyOwner
```

```
function lockMinting() external onlyOwner
```

Shell

```
constructor(address token_, address[8] memory alloc,
uint64 startTime_, address initialOwner)
Ownable(initialOwner)
function initDistribution(address nftVestingPool) external
onlyOwner
function release(uint256 id) external {
    ...
    require(msg.sender == s.beneficiary || msg.sender ==
owner(), "unauthorized");
    ...
}
```

Shell

```
constructor(address token_, address nft_, uint64
startTime_, address initialOwner) Ownable(initialOwner)
function setAllocation(uint256 tokenId, uint256 amount)
external onlyOwner
function setAllocationsBatch(uint256[] calldata tokenIds,
uint256[] calldata amounts) external onlyOwner
function finalizeAllocations() external onlyOwner
function setClaimEnabled(bool enabled) external onlyOwner
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

Team Update

The team has acknowledged that this is not a security issue and states:

The onlyOwner controls are intentional for secure setup and maintenance. They cannot change token supply or vesting once finalized. Ownership may later migrate to a multisig if needed. No exploitable risk — finding acknowledged as informational.

MUA - Metadata Update Authority

Criticality	Minor / Informative
Location	MIYINFT.sol#L3672
Status	Acknowledged

Description

The contract owner has the authority to change the metadata of the tokens. The owner may execute this by calling the `setBaseURI` function.

Shell

```
function setBaseURI(string calldata newURI) external
onlyOwner {
    _baseTokenURI = newURI;
    emit BaseURIChanged(newURI);
}
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

Team Update

The team has acknowledged that this is not a security issue and states:

The setBaseURI function is intentionally restricted to the contract owner for metadata maintenance (e.g., fixing IPFS or CDN links). It does not modify token ownership or supply. Key management and operational procedures are in place to mitigate misuse. Finding acknowledged as informational.

MT - Mints Tokens

Criticality	Minor / Informative
Location	MIYINFT.sol#L3657,3664
Status	Acknowledged

Description

The contract owner has the authority to mint tokens up to a `MAX_SUPPLY`. The owner may take advantage of it by calling the `mint` and `batchMint` functions. As a result, the contract tokens will be inflated.

```shell
function mint(address to) external onlyOwner {
    require(!mintLocked, "minting locked");
    require(_tokenIdCounter.current() < MAX_SUPPLY, "max
supply reached");
    _tokenIdCounter.increment();
    _safeMint(to, _tokenIdCounter.current());
}

function batchMint(address[] calldata toList) external
onlyOwner {
    require(!mintLocked, "minting locked");
    for (uint256 i = 0; i < toList.length; i++) {
        require(_tokenIdCounter.current() < MAX_SUPPLY,
"max supply reached");
        _tokenIdCounter.increment();
        _safeMint(toList[i], _tokenIdCounter.current());
    }
}
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

Team Update

The team has acknowledged that this is not a security issue and states:

Minting functions are intentionally restricted to the contract owner for controlled NFT distribution. All minting is capped by MAX_SUPPLY, ensuring no inflation risk. Private key management procedures mitigate misuse. Finding acknowledged as informational.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	MIYIVesting.sol#L383
Status	Acknowledged

Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand. Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

Specifically, the `_createCliffSchedule` has a misleading name and also a misleading argument, `cliffDuration`. In standard vesting logic, a cliff prevents users from claiming any tokens until the cliff period has elapsed, after which tokens vest linearly over time. In this implementation, however, once the `cliffDuration` has passed, users can claim the entire amount immediately, making the behavior more akin to a delayed release rather than a true cliff vesting schedule.

```
Shell
function _createCliffSchedule(address beneficiary, uint256
amount, uint64 start, uint64 cliffDuration
) internal {
    uint256 id = ++scheduleCount;
    schedules[id] = Schedule({totalAmount: amount,
released: 0, start: start + cliffDuration, duration: 1,
beneficiary: beneficiary, exists: true
```

```
    });
    emit ScheduleCreated(id, beneficiary, amount, start +
cliffDuration, 1);

}
```

Recommendation

It's always a good practice for the contract to contain method names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

Team Update

The team has acknowledged that this is not a security issue and states:

The _createCliffSchedule function is intentionally used for one-time token releases after a defined delay, not for linear vesting after a cliff. The naming was chosen for simplicity and internal differentiation from linear schedules. Function behavior is correct and as designed. Finding acknowledged as informational.

PSU - Potential Subtraction Underflow

Criticality	Minor / Informative
Location	MIYIVesting.sol#L450 MIYINFTVestingPool.sol#L514
Status	Acknowledged

Description

The contract subtracts two values, the second value may be greater than the first value if the contract owner misuses the configuration. As a result, the subtraction may underflow and cause the execution to revert.

Specifically, in the `MIYIVesting` the `start` time of a `Schedule` could be greater than the `timestamp` passed as input to the function.

```
Shell
uint256 elapsedDays = (timestamp - s.start) / 1 days;
```

The case is similar for the calculation of `elapsedDays` in the `MIYINFTVestingPool` contract.

```shell
uint256 elapsedDays = (uint256(timestamp) -
uint256(startTime)) / 1 days;
```

Recommendation

The team is advised to properly handle the code to avoid underflow subtractions and ensure the reliability and safety of the contract. The contract should ensure that the first value is always greater than the second value. It should add a sanity check in the setters of the variable or not allow executing the corresponding section if the condition is violated.

Team Update

The team has acknowledged that this is not a security issue and states:

In production, timestamp values are always greater than or equal to startTime, ensuring safe arithmetic. Optional sanity checks may be added for clarity, but no functional risk exists. Finding acknowledged as informational.

PUA - Potential Unauthorized Actions

Criticality	Minor / Informative
Location	MIYINFT.sol#L3331
Status	Acknowledged

Description

The contract makes an external call during the execution of the `_safeMint` method. The recipient could be a malicious contract that has an untrusted code in its fallback function. This could be used by a malicious user to perform unauthorized actions.

```
Shell
function _safeMint(address to, uint256 tokenId, bytes
memory data) internal virtual {
    _mint(to, tokenId);
    ERC721Utils.checkOnERC721Received(_msgSender(),
address(0), to, tokenId, data);
}
```

Recommendation

The team should ensure that external interaction cannot harm the protocol by using Checks-Effects-Interactions and reentrancy protection. Additionally, all state changes, supply/accounting, and access checks should be finalized before `_safeMint` runs.

Team Update

The team has acknowledged that this is not a security issue and states:

The _safeMint() implementation follows the OpenZeppelin ERC721 standard, which includes the mandatory onERC721Received() callback. All state changes and accounting occur before the external call, ensuring reentrancy safety. No unauthorized actions are possible. Finding acknowledged as informational.

RISP - Redundant Individual Struct Property

Criticality	Minor / Informative
Location	MIYINFTVestingPool.sol#L397
Status	Acknowledged

Description

The contract features a struct declaration which is composed solely of a single variable. This structure has been implemented with the intent of encapsulating individual data members. However, this practice contributes additional complexity to the codebase without enhancing its functionality or clarity. The presence of only one member within the struct suggests that the same purpose could be achieved more efficiently by directly declaring the variable, thereby avoiding the struct usage.

```
Shell
struct ClaimInfo {
    uint256 released;
}
```

Recommendation

Given that the struct declaration introduces superfluous complexity and overhead to the contract, it is recommended to eliminate such redundancies. Removing these redundant structs and directly declaring the members can streamline the codebase, enhance code readability, and lower execution costs by avoiding unnecessary struct instantiations. This adjustment not only simplifies the development and maintenance process but also optimizes performance.

Team Update

The team has acknowledged that this is not a security issue and states:

The ClaimInfo struct is intentionally retained to maintain flexibility and readability for future extensions (e.g., adding claim timestamps or claim counts). It introduces no security or performance risks. Finding acknowledged as informational.

ST - Stops Transactions

Criticality	Minor / Informative
Location	MIYICredit.sol#L809
Status	Acknowledged

Description

The contract owner has the authority to stop the sales for all users excluding the owner. The owner may take advantage of it by using the `pause` function. As a result, the contract may operate as a honeypot.

```shell
function _update(address from, address to, uint256 value)
    internal
    virtual
    override(ERC20)
{
    require(!paused(), "token paused");
    super._update(from, to, value);
}
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

Team Update

The team has acknowledged that this is not a security issue and states:

The pause() and unpause() functions are part of the standard OpenZeppelin Pausable pattern used for emergency control. They are not intended to restrict user activity maliciously. On mainnet deployment, the ownership will be managed through a multisig wallet to prevent misuse. Finding acknowledged as informational.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	MIYINFTVestingPool.sol#L531
Status	Acknowledged

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```shell
require(token.transfer(msg.sender, claimable), "transfer failed");
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

Team Update

The team has acknowledged that this is not a security issue and states:

The MIYINFTVestingPool contract will be pre-funded with 1.05B MIYI tokens by the MIYIVesting contract during initialization. After funding, the contract holds all tokens required for vesting, ensuring full sufficiency for user claims. No admin intervention is needed post-deployment. Finding acknowledged as informational.

ZD - Zero Division

Criticality	Minor / Informative
Location	MIYIVesting.sol#L452
Status	Acknowledged

Description

The contract is using variables that may be set to zero as denominators. This can lead to unpredictable and potentially harmful results, such as a transaction revert.

Specifically the `totalDays` is calculated by dividing `s.duration` by `1 days`. If `s.duration` is smaller the returned result will be zero which will cause a revert during the calculation of the releasable amount.

```
Shell
uint256 totalDays = s.duration / 1 days;
return (s.totalAmount * elapsedDays) / totalDays;
```

Recommendation

It is important to handle division by zero appropriately in the code to avoid unintended behavior and to ensure the reliability and safety of the contract. The contract should ensure that the divisor is always non-zero before performing a division operation. It should prevent the variables to be set to zero, or should not allow the execution of the corresponding statements.

Team Update

The team has acknowledged that this is not a security issue and states:

All vesting schedules are initialized with predefined non-zero durations (e.g., 2 or 4 years). The duration parameter is never user-controlled, ensuring that division by zero cannot occur in practice. Finding acknowledged as informational.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	MIYINFT.sol#L3637
Status	Acknowledged

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```shell
uint256 public immutable MAX_SUPPLY;
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

Team Update

The team has acknowledged that this is not a security issue and states:

The naming follows project-specific style to emphasize immutability and readability (MAX_SUPPLY used as a fixed cap indicator). This does not affect functionality or security. Finding acknowledged as informational.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	MIYINFTVestingPool.sol#L514,517
Status	Acknowledged

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
Shell
uint256 elapsedDays = (timestamp - s.start) / 1 days;

uint256 totalDays = uint256(VEST_DURATION) / 1 days;

return (allocation * elapsedDays) / totalDays;
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

Team Update

The team has acknowledged that this is not a security issue and states:

The current calculation uses integer-safe arithmetic with full-day precision (VEST_DURATION = 1095 days), which ensures no rounding loss in practice. Finding acknowledged as informational.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	MIYIVesting.sol#L10,92,123,223 MIYICredit.sol#L10,92,120,150,315,622,663,770,870 MIYINFTVestingPool.sol#L10,92,123,225,253,388 MIYINFT.sol#L10,38,175,204,234,399,451,482,543,1707,2458,2528,3020,3047,3479,3581,3625
Status	Acknowledged

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
Shell
pragma solidity >=0.4.16;
pragma solidity ^0.8.20;
pragma solidity ^0.8.30;

...
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

Team Update

The team has acknowledged that this is not a security issue and states:

All contracts are compiled under Solidity 0.8.30. The additional pragma directives originate from imported OpenZeppelin dependencies and do not cause conflicts. The main contracts explicitly specify a single pragma directive (^0.8.30).Finding acknowledged as informational.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	MIYIVesting.sol#L223 MIYICredit.sol#L870 MIYINFTVestingPool.sol#L388 MIYINFT.sol#L3625
Status	Acknowledged

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```shell
pragma solidity ^0.8.30;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Team Update

The team has acknowledged that this is not a security issue and states:

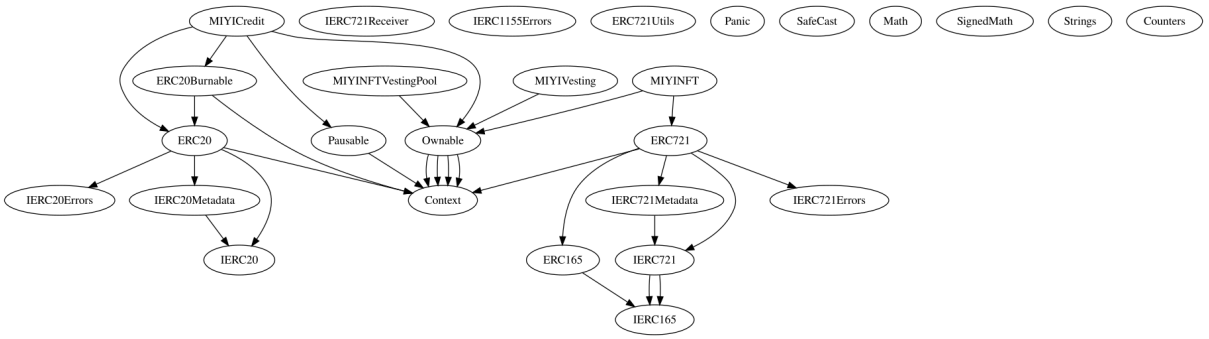
All contracts are compiled and deployed strictly with Solidity 0.8.30. The caret symbol is kept for compatibility during development, but the deployment environment enforces a fixed compiler version. Finding acknowledged as informational.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
MIYIVesting	Implementation	Ownable		
		Public	✓	Ownable
	initDistribution	External	✓	onlyOwner
	_createLinearSchedule	Internal	✓	
	_createCliffSchedule	Internal	✓	
	_createPlatformSchedule	Internal	✓	
	release	External	✓	-
	vestedAmount	Public		-
	_amountFromBP	Internal		
MIYICredit	Implementation	ERC20, ERC20Burnable, Pausable, Ownable		
		Public	✓	ERC20 Ownable
	pause	External	✓	onlyOwner
	unpause	External	✓	onlyOwner
	_update	Internal	✓	
MIYINFTVestingPool	Implementation	Ownable		

		Public	✓	Ownable
	setAllocation	External	✓	onlyOwner
	setAllocationsBatch	External	✓	onlyOwner
	finalizeAllocations	External	✓	onlyOwner
	setClaimEnabled	External	✓	onlyOwner
	vestedOf	Public		-
	claim	External	✓	-
	allocationOf	Public		-
	claimableOf	External		-
MIYINFT	Implementation	ERC721, Ownable		
		Public	✓	ERC721 Ownable
	mint	External	✓	onlyOwner
	batchMint	External	✓	onlyOwner
	_baseURI	Internal		
	setBaseURI	External	✓	onlyOwner
	lockMinting	External	✓	onlyOwner
	totalMinted	External		-

Inheritance Graph



Flow Graph

The flow graph for MIYI contracts can be found here:

myex_flow_graph.png

Summary

MIYI contract implements a token, nft and vesting mechanism. This audit investigates security issues, business logic concerns and potential improvements. The team has acknowledged all findings.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a TAC blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

A **TAC Security** Company

The Cyberscope team

cyberscope.io