Cyberscope

Audit Report

HYDT Stablecoin

June 2024

Network	BSC
Address	0x9E4FB6c5986bFd387586538554BddC1e6D54f085
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://bscscan.com/address/0x9e4fb6c5986bfd387586538554bddc1e6d54f085

Audit Updates

Initial Audit	26 Jun 2024

Source Files

Filename	SHA256
contracts/AffiliateWithdrawal.sol	48d75ab4728f9e5744b4a2fc5639ba0ca29e74a674c911fc16f93a27cbbcc3e6

Overview

The `AffiliateWithdrawal` smart contract is designed to manage and facilitate the withdrawal of tokens under specific conditions, including immediate withdrawals and those subject to vesting periods. This contract provides a structured way for users to claim and manage their token rewards, using the HYDT and HYGT tokens. The contract also incorporates security features like signature verification to ensure that withdrawals are authorized and conform to predefined terms.

ClaimWithdraw Functionality

The `claimWithdraw` function is central to the contract, allowing users to initiate the withdrawal process based on specific parameters. This function supports two main withdrawal types, immediate and vested. For immediate withdrawals, tokens are released instantly without any multipliers. However, if a vesting option is selected (e.g. 3 months or 12 months), the contract only immediately releases the HYDT tokens and records the withdrawal details, including the user’s information and vesting period, into a mapping. The HYGT tokens under vesting are locked and can only be claimed after the specified period has elapsed, leveraging different multipliers based on the duration of the vesting.

Withdraw Functionality

Once the vesting period is complete, users can claim their vested tokens through the `withdraw` function. This function is essential for users who selected a vesting option during their initial withdrawal request. It calculates the final amount of HYGT tokens to be released by applying the appropriate multiplier corresponding to the selected vesting period. The function ensures that the withdrawal conditions are met, including the passage of the vesting period, and then proceeds to mint and release the vested HYGT tokens to the user's address.

Owner Functionalities

The contract grants extensive administrative privileges to the owner, who can configure key aspects of how the contract operates. This includes setting the addresses for the HYDT and HYGT tokens, and update the signer address responsible for verifying transactions. Moreover, the owner has the authority to define and modify the vesting periods and their respective multipliers. These settings are crucial for tailoring the contract's operations to the specific needs of the token distribution strategy, making the contract adaptable to various scenarios and requirements.

Roles

Owner

The owner can interact with the following functions:

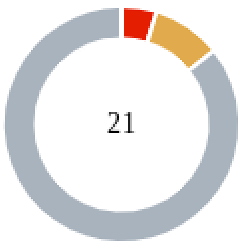
- function setSignerAddress
- function setRewardWallet
- function setHYDTAddress
- function setHYGTAddress
- function setFirstVestingPeriod
- function setSecondVestingPeriod
- function setFirstVestingmultiplier
- function setSecondVestingmultiplier

Users

The users can interact with the following functions:

- function claimWithdraw
- function withdraw

Findings Breakdown



●	Critical	1
●	Medium	2
●	Minor / Informative	18

	Severity	Unresolved	Acknowledged	Resolved	Other
●	Critical	1	0	0	0
●	Medium	2	0	0	0
●	Minor / Informative	18	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	VDM	Vesting Duration Mismatch	Unresolved
🟠	MLWB	Mint Limit Withdrawal Block	Unresolved
🟠	MCISV	Missing ChainID in Signature Validation	Unresolved
⚪	CR	Code Repetition	Unresolved
⚪	CCR	Contract Centralization Risk	Unresolved
⚪	IDI	Immutable Declaration Improvement	Unresolved
⚪	IVC	Inconsistent Validation Check	Unresolved
⚪	ISM	Inefficient State Management	Unresolved
⚪	MIT	Misleading Immutable Tokens	Unresolved
⚪	MPN	Misleading Parameter Naming	Unresolved
⚪	MEE	Missing Events Emission	Unresolved
⚪	RCC	Redundant Conditional Check	Unresolved
⚪	UVM	Unconstrained Vesting Modifications	Unresolved
⚪	UIC	Unnecessary If Check	Unresolved

●	UDE	Unused Declared Errors	Unresolved
●	URW	Unused Reward Wallet	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

VDM - Vesting Duration Mismatch

Criticality	Critical
Location	contracts/AffiliateWithdrawal.sol#L260,307
Status	Unresolved

Description

The contract is designed to validate `vestingDuration` against predetermined periods (`firstVestingPeriod` and `secondVestingPeriod`) during the withdrawal process. However, it utilizes a mapping (`userWithdrawals`) to store `vestingMonths` for each user, which is set during the initial transaction. If the predefined vesting periods are later modified, any subsequent withdrawals referencing the stored `vestingDuration` will fail due to a mismatch, as the check in `withdrawWithVesting` does not account for changes over time. This results in a scenario where users are unable to withdraw with vesting because their stored vesting duration no longer matches the updated vesting periods.

```
        withdrawWithVesting(
             _id,
             userWithdrawals[msg.sender][_id].HYGTAmount,
             userWithdrawals[msg.sender][_id].vestingMonths
         );
         ...
    function withdrawWithVesting(
         uint256 _id,
         uint256 HYGTAmount,
         uint256 vestingDuration
    ) private {
         if (
             vestingDuration != firstVestingPeriod &&
             vestingDuration != secondVestingPeriod
         ) {
             revert InvalidVestingDuration();
         }
         ...
         }
```

Recommendation

It is recommended to modify the validation logic in the `withdrawWithVesting` function to allow flexibility in checking `vestingDuration`. Instead of strictly comparing it to the current `firstVestingPeriod` and `secondVestingPeriod`, the check should consider whether the stored `vestingDuration` was valid at the time of transaction and ensure it still respects the intent of vesting rules without being affected by future changes. This approach will prevent users from being unjustly penalized for changes in vesting conditions after their transaction was recorded, thus maintaining fairness and functionality in the withdrawal process.

MLWB - Mint Limit Withdrawal Block

Criticality	Medium
Location	contracts/AffiliateWithdrawal.sol#L287
Status	Unresolved

Description

The contract is designed to mint `HYGT` tokens during the immediate withdrawal of funds, but it does not account for the maximum token supply cap of the `HYGT` token. This oversight means that when the `HYGT` token's predefined maximum supply limit is reached, any further attempts to mint tokens will fail. Consequently, this failure will also prevent the completion of the withdrawal process, as the minting step is integral to the withdrawal functionality. The absence of a check or handling mechanism for this condition can disrupt normal contract operations and adversely affect user transactions.

```
if (HYGTAmount > 0) {
        HYGT.mint(msg.sender, HYGTAmount);
}
```

Recommendation

It is recommended to consider and handle scenarios where the maximum token supply limit is reached. The contract could verify that the total supply of `HYGT` tokens has not exceeded the maximum limit before attempting to mint new tokens as part of the withdrawal process. Implementing this verification will prevent the mint function from failing and ensure that withdrawals can proceed without interruption. Additionally, appropriate error handling or alternative solutions should be designed to manage cases where the minting cannot be performed due to the max supply cap being reached.

MCISV - Missing ChainID in Signature Validation

Criticality	Medium
Location	contracts/AffiliateWithdrawal.sol#L179
Status	Unresolved

Description

The contract is currently structured to validate signatures using parameters such as the `msg.sender` , `HYDTAmount` , `HYGTAmount` , `vestingOption` , and `startTimestamp` . However, it fails to include the chainId in the parameters used for signature verification. This omission can potentially expose the contract to replay attacks, where a valid transaction on one network could be maliciously or mistakenly executed on another network, leading to unintended effects or vulnerabilities. While the contract uses `usedSignatures` to prevent the reuse of requests on the same network, this mechanism does not safeguard against cross-network replay attacks, as `usedSignatures` could potentially be unmarked on other networks.

```
    function claimWithdraw(
        uint256 HYDTAmount,
        uint256 HYGTAmount,
        uint256 vestingOption,
        uint256 startTimestamp,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        bytes32 encodedMessageHash = keccak256(
            abi.encodePacked(
                msg.sender,
                HYDTAmount,
                HYGTAmount,
                vestingOption,
                startTimestamp
            )
        );

        bytes32 signatureHash = keccak256(abi.encodePacked(v, r, s));
        if (usedSignatures[signatureHash]) {
            revert SignatureAlreadyUsed();
        }
    ...
    }
```

Recommendation

It is recommended to include the `chainId` as a parameter in the signature verification process. Incorporating chainId ensures that signatures are explicitly tied to a specific network, thus preventing transactions from being valid across different chains. This modification will significantly bolster the security of the contract against both intra-network duplicate requests and inter-network replay attacks. Enhancing the validation logic to integrate chainId will provide a robust defense against a broader range of security vulnerabilities, ensuring the integrity and network-specific execution of the contract.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L318
Status	Unresolved

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.

```
if (vestingDuration == firstVestingPeriod) {
    HYGTAmount = ((HYGTAmount * (firstVestingmultiplier)) /
        denominator);
    HYGT.mint(msg.sender, HYGTAmount);
    userWithdrawals[msg.sender][_id].HYGTAmount = HYGTAmount;
    userWithdrawals[msg.sender][_id].fundsWithdrawn = true;
} else {
    HYGTAmount = ((HYGTAmount * (secondVestingmultiplier)) /
        denominator);
    HYGT.mint(msg.sender, HYGTAmount);
    userWithdrawals[msg.sender][_id].HYGTAmount = HYGTAmount;
    userWithdrawals[msg.sender][_id].fundsWithdrawn = true;
}
```

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.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L84,104,124,150
Status	Unresolved

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.

Specifically the owner has the capability to modify key configurations such as the `signerAddress`, the addresses of the `HYDT` and `HYGT` tokens, as well as the duration and multipliers for vesting periods.

```
   function setSignerAddress(address _addr) public onlyOwner {
        signerAddress = _addr;
        emit SignerAddressUpdated(_addr);
    }

    function setHYDTAddress(IHYDT _addr) public onlyOwner {
        HYDT = _addr;
        emit HYDTAddressUpdated(_addr);
    }

    function setHYGTAddress(IHYGT _addr) public onlyOwner {
        HYGT = _addr;
        emit HYGTAddressUpdated(_addr);
    }

    function setFirstVestingPeriod(uint256 _period) public onlyOwner {
        if (_period == firstVestingPeriod) {
            revert AlreadySameValue();
        }
        firstVestingPeriod = _period;
        emit FirstVestingPeriodUpdated(_period);
    }

    function setSecondVestingPeriod(uint256 _period) public onlyOwner {
        ...
    }

    function setFirstVestingmultiplier(uint256 _multiplier) public
onlyOwner {
        if (_multiplier == firstVestingmultiplier) {
            revert AlreadySameValue();
        }
        firstVestingmultiplier = _multiplier;
        emit FirstVestingmultiplierUpdated(_multiplier);
    }


    function setSecondVestingmultiplier(uint256 _multiplier) public
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.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L75,76
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
month
denominator
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

IVC - Inconsistent Validation Check

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L279
Status	Unresolved

Description

The contract includes a validation check to ensure that the `HYGTAmount` is greater than zero, which is essential for maintaining the integrity of transactions involving this parameter. However, it fails to apply a similar validation for `HYDTAmount`. This omission allows `HYDTAmount` to be zero, which may not align with the intended logic of the contract. This inconsistency in validation practices can lead to unintended behavior or errors, undermining the contract's reliability.

```
if (HYGTAmount <= 0) {
    revert InvalidWithdrawalAmount();
}
```

Recommendation

It is recommended to add an additional check to prevent the `HYDTAmount` from being zero, ensuring that the validation logic is consistent across different transaction amounts. This adjustment will enhance the contract's robustness by safeguarding against invalid inputs and ensuring uniform enforcement of transaction rules.

ISM - Inefficient State Management

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L358,394
Status	Unresolved

Description

The contract is currently handling updates to user withdrawal records by assigning each property individually. This approach involves multiple state changes for each property of a single withdrawal record. Such a process is inefficient in terms of gas consumption, as each separate state change incurs additional computational cost. Moreover, this method increases the complexity of transaction handling within the blockchain environment, potentially leading to higher execution times and greater load.

```
userWithdrawals[user][_id].HYDTAmount = HYDTAmount;
userWithdrawals[user][_id].HYGTAmount = HYGTAmount;
userWithdrawals[user][_id].chosenVestingDuration = block.timestamp;
userWithdrawals[user][_id].vestingMonths = vestingOption;
userWithdrawals[user][_id].fundsWithdrawn = fundWidraw;
userWithdrawals[user][_id].dataAvailable = true;
...
userWithdrawals[user][_id].HYDTAmount = HYDTAmount;
userWithdrawals[user][_id].HYGTAmount = HYGTAmount;
userWithdrawals[user][_id].chosenVestingDuration =
    startTime +
    (month * vestingOption);
userWithdrawals[user][_id].vestingMonths = vestingOption;
userWithdrawals[user][_id].fundsWithdrawn = fundWidraw;
userWithdrawals[user][_id].dataAvailable = true;
```

Recommendation

It is recommended to refactor the contract to utilize a struct for encapsulating all attributes related to a user's withdrawal record. By bundling these attributes into a single struct and assigning it in one operation, the contract can significantly reduce the number of state changes. This single transaction update not only minimizes the gas costs associated with multiple state changes but also simplifies the code, enhancing both performance and maintainability. Implementing this change will lead to more efficient contract operations and a reduction in transaction fees for users.

MIT - Misleading Immutable Tokens

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L105,115
Status	Unresolved

Description

The contract is designed with functions `setHYDTAddress` and `setHYGTAddress` that allow the owner to modify the addresses of the HYDT and HYGT tokens. However, the use of uppercase letters for these token addresses suggests that they are immutable variables. This discrepancy between the naming convention and the actual functionality is misleading and could cause confusion regarding the mutability of these addresses.

```
    function setHYDTAddress(IHYDT _addr) public onlyOwner {
        HYDT = _addr;
        emit HYDTAddressUpdated(_addr);
    }


    function setHYGTAddress(IHYGT _addr) public onlyOwner {
        HYGT = _addr;
        emit HYGTAddressUpdated(_addr);
    }
```

Recommendation

It is recommended to either rename the token address variables to follow a naming convention that does not imply immutability or to make these variables truly immutable by removing the setter functions. This will ensure clarity and prevent any misconceptions about the contract’s behavior.

MPN - Misleading Parameter Naming

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L207,273,290,345
Status	Unresolved

Description

The contract is found to inconsistently name a boolean variable across different functions, which introduces ambiguity and potential misinterpretation of its intended usage. Specifically, the boolean variable is passed as `amountWithdraw` into the `withdrawImmediately` function, but within the `processWithdrawalInfo` function, it is referred to as `fundWidraw`. This shift in naming for the same boolean variable is misleading. Additionally, the name `amountWithdraw` misleadingly suggests it might be a numerical value representing an amount, rather than a boolean. This inconsistency could lead to errors in understanding the contract's logic or in further development phases.

```
withdrawImmediately(HYDTAmount, HYGTAmount, vestingOption, true);
...
    function withdrawImmediately(
        uint256 HYDTAmount,
        uint256 HYGTAmount,
        uint256 vestingDuration,
        bool amountWithdraw
        ...
        if (HYGTAmount != 0) {
            processWithdrawalInfo(
                msg.sender,
                HYDTAmount,
                HYGTAmount,
                vestingDuration,
                amountWithdraw
            );

        }
        ...
    function processWithdrawalInfo(
        address user,
        uint256 HYDTAmount,
        uint256 HYGTAmount,
        uint256 vestingOption,
        bool fundWidraw
```

Recommendation

It is recommended to rename the `amountWithdraw` parameter to a name that clearly indicates its boolean nature, such as isImmediateWithdrawal, or to maintain consistent naming across all functions where this variable is used, such as `fundWidraw`. This will enhance code readability and reduce the risk of confusion in the functionality of the contract. Maintaining consistent, descriptive, and type-indicative naming conventions is crucial for the maintainability and clarity of the code.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L307
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
    function withdrawWithVesting(
        uint256 _id,
        uint256 HYGTAmount,
        uint256 vestingDuration
    ) private {
        if (
            vestingDuration != firstVestingPeriod &&
            vestingDuration != secondVestingPeriod
        ) {
            revert InvalidVestingDuration();
        }
        ...
        }
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

RCC - Redundant Conditional Check

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L290
Status	Unresolved

Description

The contract is structured to include an initial `if` statement that reverts the transaction if the `HYGTAmount` is zero or negative, which is a check to ensure valid transaction amounts. Subsequently, there is another `if` statement checking if `HYGTAmount` is not zero before calling the `processWithdrawalInfo` function. Since the first condition already ensures that `HYGTAmount` cannot be zero, the second condition becomes inherently redundant. This redundancy does not contribute to functional enhancement and may lead to confusion or unnecessary complexity in the contract's logic.

```
if (HYGTAmount <= 0) {
    revert InvalidWithdrawalAmount();
}
...
if (HYGTAmount != 0) {
    processWithdrawalInfo(
        msg.sender,
        HYDTAmount,
        HYGTAmount,
        vestingDuration,
        amountWithdraw
    );
}
```

Recommendation

It is recommended to remove the second `if` statement since it will always evaluate to true in the current implementation, given the prior validation. Removing this redundant check will simplify the contract code, reduce unnecessary computational checks, and clarify the execution path, thereby optimizing the contract's performance and readability.

UVM - Unconstrained Vesting Modifications

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L
Status	Unresolved

Description

The contract is designed with functions that enable the modification of parameters within the vesting mechanism. These functions allow the owner to independently set the vesting periods and multipliers. However, this independence can lead to inconsistencies and incorrect calculations, as there are no constraints ensuring that multipliers correspond appropriately to their respective vesting periods. This can affect the accuracy and fairness of the token distribution process, potentially leading to erroneous token minting and user rewards.

```
   function setFirstVestingPeriod(uint256 _period) public onlyOwner {
        if (_period == firstVestingPeriod) {
            revert AlreadySameValue();
        }
        firstVestingPeriod = _period;
        emit FirstVestingPeriodUpdated(_period);
    }

    function setSecondVestingPeriod(uint256 _period) public onlyOwner {
        if (_period == secondVestingPeriod) {
            revert AlreadySameValue();
        }
        secondVestingPeriod = _period;
        emit SecondVestingPeriodUpdated(_period);
    }

    function setFirstVestingmultiplier(uint256 _multiplier) public
onlyOwner {
        if (_multiplier == firstVestingmultiplier) {
            revert AlreadySameValue();
        }
        firstVestingmultiplier = _multiplier;
        emit FirstVestingmultiplierUpdated(_multiplier);
    }

    function setSecondVestingmultiplier(uint256 _multiplier) public
onlyOwner {
        if (_multiplier == secondVestingmultiplier) {
            revert AlreadySameValue();
        }
        secondVestingmultiplier = _multiplier;
        emit SecondVestingmultiplierUpdated(_multiplier);
    }
```

Recommendation

It is recommended to incorporate checks that enforce strict limits on the modification of vesting parameters. Specifically, ensure that multiplier values are appropriately correlated with their respective vesting periods. This can involve setting predefined ranges or logical constraints that bind multipliers to specific periods, thereby maintaining the integrity and intended function of the vesting mechanism.

UIC - Unnecessary If Check

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L205,273,345
Status	Unresolved

Description

The contract is designed to validate the `vestingOption` parameter in the `processWithdrawalInfo` function by checking if it is not zero. However, this validation is redundant as the `vestingOption` is already checked in the preceding `claimWithdraw` function, which directs the flow either to an immediate withdrawal or further processing based on whether `vestingOption` is zero. Since `processWithdrawalInfo` is invoked conditionally when `vestingOption` is not zero, the additional check within this function is unnecessary and does not impact the flow of execution or enhance security, thereby merely adding to the complexity and computational cost of the contract.

```
   function claimWithdraw(
        uint256 HYDTAmount,
        uint256 HYGTAmount,
        uint256 vestingOption,
        uint256 startTimestamp,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
    ...
        if (vestingOption == 0) {
            // Immediate withdrawal
            withdrawImmediately(HYDTAmount, HYGTAmount, vestingOption,
true);
        }
    }

        ...
    function withdrawImmediately(
        uint256 HYDTAmount,
        uint256 HYGTAmount,
        uint256 vestingDuration,
        bool amountWithdraw
    ) private {
        ...
        if (HYGTAmount != 0) {
            processWithdrawalInfo(
            ...
            }

    function processWithdrawalInfo(
        address user,
        uint256 HYDTAmount,
        uint256 HYGTAmount,
        uint256 vestingOption,
        bool fundWidraw
    ) private {
        uint256 _id = userTotalWithdrawals[msg.sender]++;

        if (vestingOption != 0) {
            revert InvalidVestingDuration();
        }
    ...
    }
```

Recommendation

It is recommended to remove the redundant check for `vestingOption` in the `processWithdrawalInfo` function, as it does not provide additional safety or functional benefit. This simplification will reduce the computational overhead and streamline the contract's logic, making the code more efficient and easier to understand. Ensuring that checks are not unnecessarily duplicated also aids in maintaining cleaner, more maintainable code.

UDE - Unused Declared Errors

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L62
Status	Unresolved

Description

The contract is declaring the errors `InsufficientHYDTallowance` and `InsufficientHYGTallowance` which are not utilized within the contract's functions. Declaring errors without using them can lead to confusion, increase the contract's bytecode size unnecessarily, and potentially mislead future developers or auditors about the contract's error handling capabilities.

```
error InsufficientHYDTallowance();
error InsufficientHYGTallowance();
```

Recommendation

It is recommended to either utilize these declared errors within the appropriate functions to handle relevant conditions or remove these unused error declarations from the contract to maintain clarity and optimize the bytecode size. This will ensure that the contract remains efficient and easier to understand and maintain.

URW - Unused Reward Wallet

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L94
Status	Unresolved

Description

The contract is configured with a `rewardWallet` variable that can be set via the `setRewardWallet` function. However, the `rewardWallet` is not utilized within the contract's code for any operations or logic. This could indicate either a missing implementation or an unnecessary variable, which might confuse developers and users who interact with the contract.

```
    function setRewardWallet(address _addr) public onlyOwner {
        rewardWallet = _addr;
        emit RewardWalletAddressUpdated(_addr);
    }
```

Recommendation

It is recommended to either remove the `rewardWallet` variable and its associated function if it is not required or to implement the necessary logic that utilizes this wallet as intended. This will ensure clarity and purpose in the contract's code, reducing potential misunderstandings and maintaining clean, efficient code.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L15,16,84,94,104,114,124,137,150,163,180,181,243,274,275,308,309,347,348,379,380,423,424
Status	Unresolved

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.

```
IHYDT public HYDT
IHYGT public HYGT
address _addr
IHYDT _addr
IHYGT _addr
uint256 _period
uint256 _multiplier
uint256 HYDTAmount
uint256 HYGTAmount
uint256 _id
```

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/v0.8.17/style-guide.html#naming-convention.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L421,442
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function recoverSigner(
        address user,
        uint256 HYDTAmount,
        uint256 HYGTAmount,
        uint256 vestingOption,
        uint256 timestamp,
...
                vestingOption,
                timestamp
            )
        );

        return messageHash.recover(signature);
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L319,327
Status	Unresolved

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.

```
HYGTAmount = ((HYGTAmount * (firstVestingmultiplier)) /
                    denominator)
HYGTAmount = ((HYGTAmount * (secondVestingmultiplier)) /
                    denominator)
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L70,85,95
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
signerAddress = _signerAddress
signerAddress = _addr
rewardWallet = _addr
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/AffiliateWithdrawal.sol#L2
Status	Unresolved

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.

```
pragma solidity ^0.8.19;
```

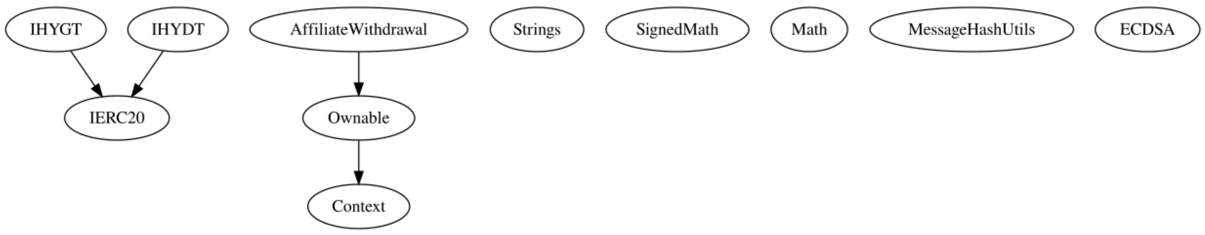
Recommendation

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.

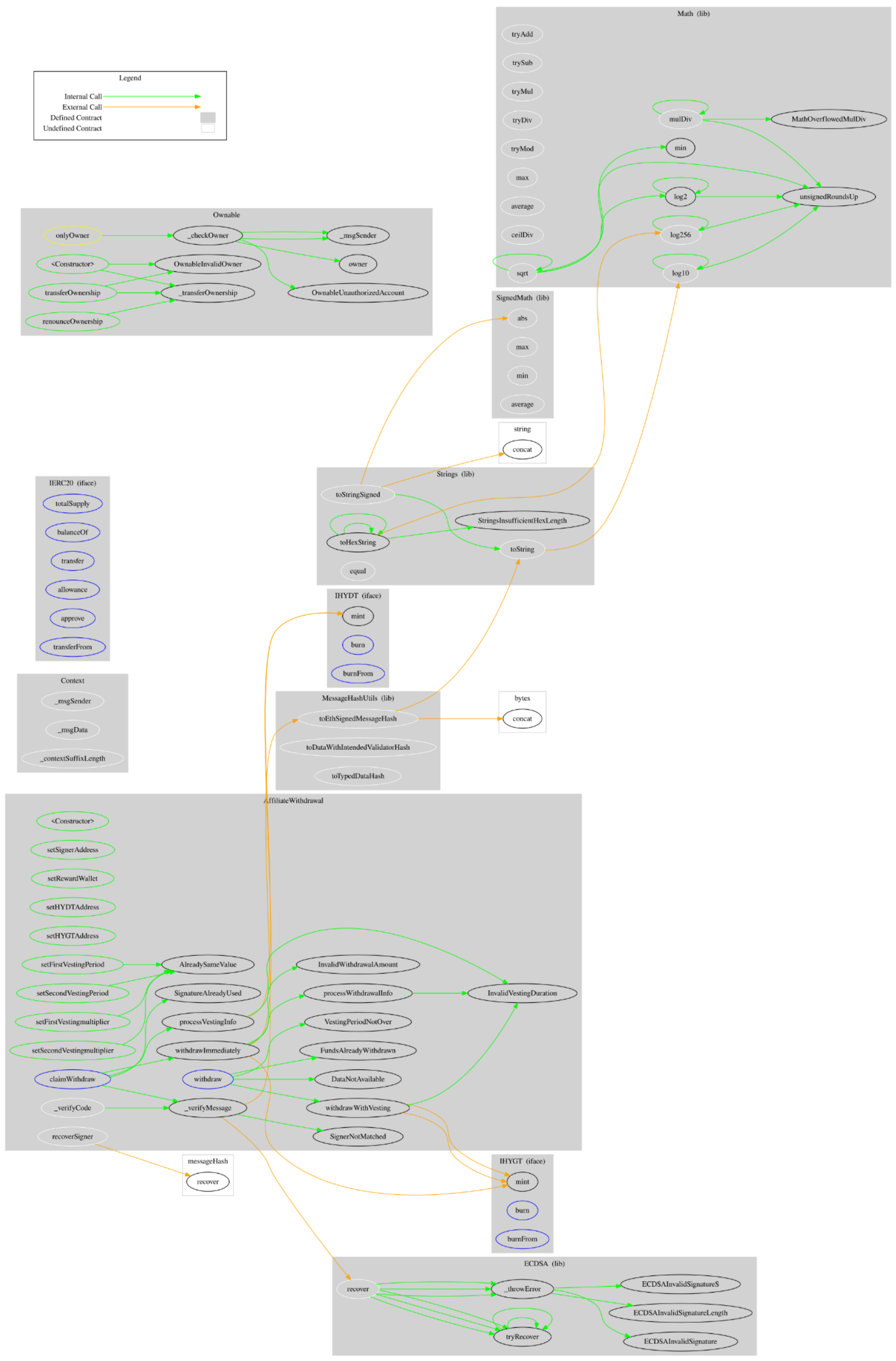
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
AffiliateWithdrawal	Implementation	Ownable		
		Public	✓	Ownable
	setSignerAddress	Public	✓	onlyOwner
	setRewardWallet	Public	✓	onlyOwner
	setHYDTAddress	Public	✓	onlyOwner
	setHYGTAddress	Public	✓	onlyOwner
	setFirstVestingPeriod	Public	✓	onlyOwner
	setSecondVestingPeriod	Public	✓	onlyOwner
	setFirstVestingmultiplier	Public	✓	onlyOwner
	setSecondVestingmultiplier	Public	✓	onlyOwner
	claimWithdraw	External	✓	-
	_verifyMessage	Private		
	withdraw	External	✓	-
	withdrawImmediately	Private	✓	
	withdrawWithVesting	Private	✓	
	processWithdrawalInfo	Private	✓	
	processVestingInfo	Private	✓	
	recoverSigner	Internal		
	_verifyCode	Internal		

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Math (lib)
tryAdd
trySub
tryMul
tryDiv
tryMod
max
average
ceilDiv
sqrt
mulDiv
MathOverflowedMulDiv
min
log2
unsignedRoundsUp
log256
log10
Ownable
onlyOwner
_checkOwner
_msgSender
<Constructor>
OwnableInvalidOwner
owner
transferOwnership
_transferOwnership
OwnableUnauthorizedAccount
renounceOwnership
SignedMath (lib)
abs
max
min
average
string
concat
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Strings (lib)
toStringSigned
toHexString
StringsInsufficientHexLength
toString
equal
IHYDT (iface)
mint
burn
burnFrom
Context
_msgSender
_msgData
_contextSuffixLength
MessageHashUtils (lib)
toEthSignedMessageHash
toDataWithIntendedValidatorHash
toTypedDataHash
bytes
concat
AffiliateWithdrawal
<Constructor>
setSignerAddress
setRewardWallet
setHYDTAddress
setHYGTAddress
setFirstVestingPeriod
AlreadySameValue
InvalidWithdrawalAmount
setSecondVestingPeriod
SignatureAlreadyUsed
processWithdrawalInfo
InvalidVestingDuration
setFirstVestingmultiplier
processVestingInfo
VestingPeriodNotOver
setSecondVestingmultiplier
withdrawImmediately
FundsAlreadyWithdrawn
claimWithdraw
withdraw
DataNotAvailable
_verifyCode
_verifyMessage
withdrawWithVesting
recoverSigner
SignerNotMatched
messageHash
recover
IHYGT (iface)
mint
burn
burnFrom
ECDSA (lib)
recover
_throwError
ECDSAInvalidSignatureS
ECDSAInvalidSignatureLength
tryRecover
ECDSAInvalidSignature

Summary

The AffiliateWithdrawal contract implements a withdrawal mechanism for the HYDT and HYGT tokens, facilitating user withdrawals with options for immediate or vested disbursements. This audit investigates security issues, business logic concerns, and potential improvements to ensure the contract's functionality and user protection.

Disclaimer

About Cyberscope

Cyberscope is a 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.

The Cyberscope team

https://www.cyberscope.io