YAYToken&Vesting

Smart Contract Audit Report
Prepared for Yay Games

Date Issued:	Sep 6, 2021
Project ID:	AUDIT2021016
Version:	v1.0
Confidentiality Level:	Public

Report Information

Project ID	AUDIT2021016
Version	v1.0
Client	Yay Games
Project	YAYToken&Vesting
Auditor(s)	Suvicha Buakhom Patipon Suwanbol
Author	Suvicha Buakhom
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Sep 6, 2021	Full report	Suvicha Buakhom

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Yay Games, Inspex team conducted an audit to verify the security posture of the YAYToken&Vesting smart contracts between Aug 24, 2021 and Aug 25, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of YAYToken&Vesting smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

The issues identified are mostly related to the centralized design of the smart contracts, providing the contract owner with high privilege, including the ability to pause the transfer of the token or blacklist and burn tokens from specific addresses. The project team has resolved most issues, but decided to keep the pausing and blacklisting mechanism in the contract. Therefore, to use the platform, the users need to know that these mechanisms exist, and have to trust that the platform owner will not use those mechanisms for a malicious purpose.

1.1. Audit Result

In the initial audit, Inspex found 2 high, 2 medium, 1 very low, and 2 info-severity issues. With the project team's response, 1 high, 1 medium, 1 very low, and 2 info-severity issues were resolved in the reassessment, while 1 high and 1 medium-severity issues were acknowledged by the team. In the long run, Inspex suggests resolving all issues found in this report as soon as possible to reduce the risk and improve the security level of smart contracts.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Yay Games is a fully decentralized Gaming Marketplace designed for Gamers, Traders, and Farmers. The protocol participants can earn crypto from in-game mining, farming, and trading.

YAY Token is a standard BEP20 token with blacklisting and pausing mechanisms. The token is a limited supply to 1 billion $YAY.

Vesting is a mechanism to distribute tokens to users. On Vesting, users can claim tokens in portions that are allocated to each of them.

Scope Information:

Project Name	YAYToken&Vesting
Website	https://www.yay.games/
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Aug 24, 2021 - Aug 25, 2021
Reassessment Date	Sep 3, 2021

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 2ce693244131dc96822c7b949ba93a0de332016c)

Contract	Location (URL)
BEP20YAY	https://github.com/YAY-Games/token-contract/blob/2ce6932441/contracts/BEP20YAY.sol

Initial Audit: (Commit: ee8a98770848c566410bdebb269a69131c22de16)

Contract	Location (URL)
YayVesting	https://github.com/YAY-Games/vesting-contract/blob/ee8a987708/contracts/YayVesting.sol

Reassessment: (Commit: 1af586553e063c1fac12924de72e037bc4b3f34e)

Contract	Location (URL)
BEP20YAY	https://github.com/YAY-Games/token-contract/blob/1af586553e/contracts/BEP20YAY.sol

Reassessment: (Commit: e58e873f8a414c1f8bbf64d1cb9c33b83bcf0bea)

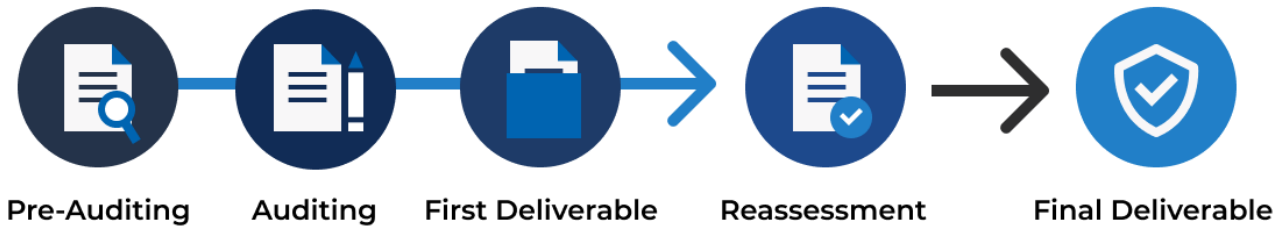
Contract	Location (URL)
YayVesting	https://github.com/YAY-Games/vesting-contract/blob/e58e873f8a/contracts/YayVesting.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they are inherited from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Upgradable Without Timelock
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation

Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

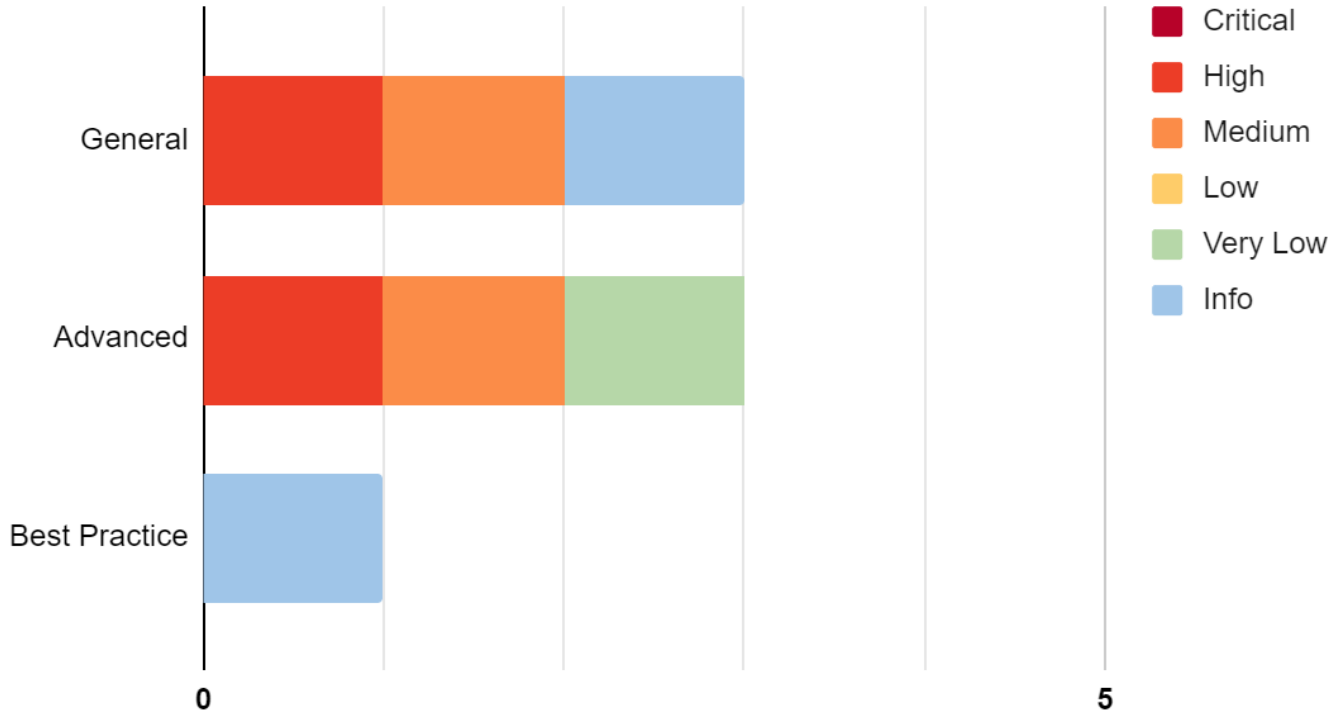
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	**Very Low**	**Low**	**Medium**
Medium	**Low**	**Medium**	**High**
High	**Medium**	**High**	**Critical**

4. Summary of Findings

From the assessments, Inspex has found 7 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Design Flaw in Minting Cap Validation	Advanced	High	Resolved
IDX-002	Unusable Tokens via Blacklisting	General	High	Acknowledged
IDX-003	Untransferable Tokens via Contract Pausing	General	Medium	Acknowledged
IDX-004	Token Draining by emergencyWithdrawal()	Advanced	Medium	Resolved
IDX-005	Insufficient Logging for Privileged Functions	Advanced	Very Low	Resolved
IDX-006	Unsafe Token Transfer	General	Info	Resolved
IDX-007	Improper Function Visibility	Best Practice	Info	Resolved

5. Detailed Findings Information

5.1. Design Flaw in Minting Cap Validation

ID	IDX-001
Target	BEP20YAY
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The contract owner can mint the $YAY that get burned from the users with the same amount and arbitrarily steal the tokens with no limitation by exploiting **IDX-002 Unusable Tokens via Blacklisting**. **Likelihood:** Medium There is no mechanism to prevent the owner from executing these actions.
Status	Resolved Yay Games team has resolved this issue in commit `1499d04af27b92ddaad39def4ab6e272708d947c` by changing the burn mechanism logic to transfer to `address(0)`, so the total supply of the $YAY will be not decreased.

5.1.1. Description

In `BEP20YAY` contract ($YAY), the `mint()` function allows the contract owner to mint any amount of $YAY freely as follows.

BEP20YAY.sol

```
615 function mint(uint256 amount) external onlyOwner returns(bool) {
616     _mint(_msgSender(), amount);
617     return true;
618 }
```

The only limitation is that the total minted tokens must not increase the total supply to exceed the total supply cap (`_cap`) as shown below:

BEP20YAY.sol

```
696 function _mint(address account, uint256 amount) internal {
697     require(account != address(0), "BEP20: mint to the zero address");
698     require(_totalSupply.add(amount) <= _cap, "BEP20: cap exceeded");
699
700     _beforeTokenTransfer(address(0), account);
```

```
701
702     _totalSupply = _totalSupply.add(amount);
703     _balances[account] = _balances[account].add(amount);
704     emit Transfer(address(0), account, amount);
705 }
```

However, there is a mechanism to reduce the total supply of the $YAY, allowing additional minting of the $YAY. This can be achieved from 3 approaches, which are the `burn()` function, the `burnFrom()` function, and the `burnBlackFunds()` function. All of these functions call `_burn()` function to reduce the $YAY's total supply.

BEP20YAY.sol

```
718 function _burn(address account, uint256 amount) internal {
719     require(account != address(0), "BEP20: burn from the zero address");
720
721     _beforeTokenTransfer(account, address(0));
722
    _balances[account] = _balances[account].sub(amount, "BEP20: burn amount
723 exceeds balance");
724     _totalSupply = _totalSupply.sub(amount);
725     emit Transfer(account, address(0), amount);
726 }
```

For example, the `burnFrom()` function burns the `amount` of the $YAY according to the given allowance of `_msgSender()`.

BEP20YAY.sol

```
656 function burnFrom(address account, uint256 amount) external {
657     uint256 decreasedAllowance = _allowances[account][_msgSender()].sub(amount,
    "BEP20: burn amount exceeds allowance");
658     _approve(account, _msgSender(), decreasedAllowance);
659     _burn(account, amount);
660 }
```

Nevertheless, by taking the advantage of **IDX-002 Unusable Tokens via Blacklisting** issue, the contract owner can burn $YAY of any user in order to reduce the total supply, bypassing the only limitation of minting $YAY.

BEP20YAY.sol

```
645 function burnBlackFunds(address target, uint256 amount) public onlyOwner {
646     require(isBlacklisted(target));
647     _burn(target, amount);
648 }
```

BEP20YAY.sol

```
718 function _burn(address account, uint256 amount) internal {
719     require(account != address(0), "BEP20: burn from the zero address");
720
721     _beforeTokenTransfer(account, address(0));
722
723     _balances[account] = _balances[account].sub(amount, "BEP20: burn amount
    exceeds balance");
724     _totalSupply = _totalSupply.sub(amount);
725     emit Transfer(account, address(0), amount);
726 }
```

5.1.2. Remediation

Inspex suggests removing the burn mechanism from the contract. However, if the burn mechanism is needed, it is suggested that the burn mechanism must not reduce the $YAY's total supply.

This can be achieved by transferring the burned tokens to the dead address.

5.2. Unusable Tokens via Blacklisting

ID	IDX-002
Target	BEP20YAY
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity: High** **Impact: High** Specific users cannot use the tokens freely due to getting their transfer blocked or getting their token burned. Moreover, by taking advantage of the **IDX-001 Design Flaw in Minting Cap Validation** issue, the contract owner can steal the $YAY from any user. **Likelihood: Medium** It is very likely that the platform will blacklist a specific address to gain the advantages since blacklisting can prevent the user from selling $YAY as an example.
Status	**Acknowledged** Yay Games team has acknowledged this issue.

5.2.1. Description

The $YAY token is implemented as an ERC20 standard token. It includes the blacklist mechanism to prevent all token transfers and burn the tokens when the contract owners decide to prevent further actions for a specific address.

After a specific address gets blacklisted, all of the token transfers from that address will be blocked. This is because the `_transfer()` function will be executed whenever the tokens are transferred.

BEP20YAY.sol

```
676 function _transfer(address sender, address recipient, uint256 amount) internal
    {
677     require(sender != address(0), "BEP20: transfer from the zero address");
678     require(recipient != address(0), "BEP20: transfer to the zero address");
679
680     _beforeTokenTransfer(sender, recipient);
681
682     _balances[sender] = _balances[sender].sub(amount, "BEP20: transfer amount
    exceeds balance");
683     _balances[recipient] = _balances[recipient].add(amount);
684     emit Transfer(sender, recipient, amount);
685 }
```

The `_transfer()` function calls the `_beforeTokenTransfer()` function, which will revert the transaction if the $YAY token is paused as long as the caller is not the contract owner (`owner()`).

BEP20YAY.sol

```
762 function _beforeTokenTransfer(address from, address to) internal view {
763     if (_msgSender() != owner()) {
764         require(!paused(), "BEP20: token transfer while paused");
765         require(!isBlacklisted(from) && !isBlacklisted(to), "BEP20: account is
    blacklisted");
766     }
767 }
```

Furthermore, the tokens of the blacklisted address could potentially get burned by the contract owner.

In the `BEP20YAY` contract, the `burnBlackFunds()` function will burn the tokens from the `target` address.

BEP20YAY.sol

```
645 function burnBlackFunds(address target, uint256 amount) public onlyOwner {
646     require(isBlacklisted(target));
647     _burn(target, amount);
648 }
```

The `isBlacklisted(target)` function will validate that the `target` address is in the blacklist or not.

BEP20YAY.sol

```
427 function isBlacklisted(address account) public view returns (bool) {
428     return blacklisted[account];
429 }
```

However, the contract owner can freely add any address to the blacklist by using the `blacklist()` function.

BEP20YAY.sol

```
434 function blacklist(address account) external onlyOwner {
435     require(!blacklisted[account], "BEP20: blacklisted");
436     blacklisted[account] = true;
437     emit Blacklisted(account);
438 }
```

Hence, the contract owner can remove the tokens from any address without the user's permission. As a result, by taking advantage of the **IDX-001 Design Flaw in Minting Cap Validation** issue, the contract owner can steal the $YAY from any user.

5.2.2. Remediation

Inspex suggests removing the blacklist mechanism since it is a purely centralized control design, which should not exist in the decentralized finance as it gives the absolute power to the owner.

However, if the blacklist mechanism is needed, it is suggested that the platform should implement a community-run governance to control the use of these functions.

5.3. Untransferable Tokens via Contract Pausing

ID	IDX-003
Target	BEP20YAY
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Medium **Impact:** Medium The platform can gain advantages from the users by controlling token transfer flows, resulting in disadvantages for the users. **Likelihood:** Medium It is very likely that the contract owner will pause the token transfer. However, there is low motivation for the contract owner to do this since pausing the token transfer means that no one can transfer $YAY, including the router contract, except the contract owner. As a result, it is a non-profitable scenario.
Status	**Acknowledged** Yay Games team has acknowledged this issue.

5.3.1. Description

The $YAY token is implemented as an `ERC20` standard token. It includes the pause mechanism to prevent all token transfers when the contract owner decides to pause it.

Whenever the tokens are transferred, the `_transfer()` function will be executed as follows:

BEP20YAY.sol

```
676 function _transfer(address sender, address recipient, uint256 amount) internal
    {
677     require(sender != address(0), "BEP20: transfer from the zero address");
678     require(recipient != address(0), "BEP20: transfer to the zero address");
679
680     _beforeTokenTransfer(sender, recipient);
681
682     _balances[sender] = _balances[sender].sub(amount, "BEP20: transfer amount
    exceeds balance");
683     _balances[recipient] = _balances[recipient].add(amount);
684     emit Transfer(sender, recipient, amount);
685 }
```

It calls the `_beforeTokenTransfer()` function, which will revert the transaction if the $YAY token is paused as long as the caller is not the contract owner.

BEP20YAY.sol

```
762 function _beforeTokenTransfer(address from, address to) internal view {
763     if (_msgSender() != owner()) {
764         require(!paused(), "BEP20: token transfer while paused");
765         require(!isBlacklisted(from) && !isBlacklisted(to), "BEP20: account is
    blacklisted");
766     }
767 }
```

5.3.2. Remediation

Inspex suggests removing the pausing mechanism from the contract to ensure that the tokens can be transferred freely.

However, if the pausing mechanism is needed to prevent any unexpected issue on the platform, it is recommended to implement this feature to the other contract logic instead since the users' tokens should be transferred freely at any cost.

5.4. Token Draining by emergencyWithdrawal()

ID	IDX-004
Target	YayVesting
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The tokens that are deposited can be drained by the contract owner. **Likelihood:** Medium There is no restriction to prevent the contract owner from executing this action.
Status	Resolved Yay Games team has resolved this issue by removing the `emergencyWithdrawal()` function as suggested in commit `6af932cdd95303a8c5bd19f140199a78e329bd3c`.

5.4.1. Description

The `YayVesting` contract can be used for distributing the funds to specific groups of users. Unfortunately, it is possible to take the deposited tokens out from the `YayVesting` contract with the `emergencyWithdrawal()` function by the contract owner directly as shown in the following source code:

YayVesting.sol

```
89  function emergencyWithdrawal(uint256 amount) external onlyOwner returns(bool) {
90      require(amount > 0, "YayVesting: amount must be greater than 0");
91      IERC20(token).transfer(msg.sender, amount);
92      return true;
93  }
```

5.4.2. Remediation

The allocated tokens should be withdrawn only to those who have already been allocated. Inspex suggests removing the `emergencyWithdrawal()` function from the `YayVesting` contract to make the contract owner unable to withdraw the deposited tokens of the users.

5.5. Insufficient Logging for Privileged Functions

ID	IDX-005
Target	YayVesting
Category	Advanced Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved Yay Games team has resolved this issue by removing the `emergencyWithdrawal()` function in commit `6af932cdd95303a8c5bd19f140199a78e329bd3c`.

5.5.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts to the platform.

In the `YayVesting` contract, the owner can withdraw all tokens (`token`) by executing `emergencyWithdrawal()`, and no events are emitted.

YayVesting.sol

```
89 function emergencyWithdrawal(uint256 amount) external onlyOwner returns(bool) {
90     require(amount > 0, "YayVesting: amount must be greater than 0");
91     IERC20(token).transfer(msg.sender, amount);
92     return true;
93 }
```

5.5.2. Remediation

Inspex suggests emitting events for the `emergencyWithdrawal()` function, for example:

YayVesting.sol

```
88 event EmergencyWithdrawal(uint256 amount);
89 function emergencyWithdrawal(uint256 amount) external onlyOwner returns(bool) {
90     require(amount > 0, "YayVesting: amount must be greater than 0");
91     IERC20(token).transfer(msg.sender, amount);
92     emit EmergencyWithdrawal(amount);
```

```
93         return true;
94     }
```

5.6. Unsafe Token Transfer

ID	IDX-006
Target	YayVesting
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Yay Games team has resolved this issue as suggested in commit `6af932cdd95303a8c5bd19f140199a78e329bd3c`.

5.6.1. Description

The `claim()` function allows the specific users to acquire tokens (`token`) in the `YayVesting` contract. The `token` could be improperly implemented since there is no guarantee that the `token` comes with handling an insufficient amount transfer.

This allows the execution of `transfer()` function without reverting when the invalid transfer amount occurs, causing significant damage to the smart contract if the token is insufficient.

For example, the user claims the tokens with an amount that exceeds the token amount in the contract, but there is no reverting in the `transfer()` function. The claimed tokens will be registered as claimed successfully (in line number 105) without having any token transferred to the user.

YayVesting.sol

```
 99 function claim(uint256 _category, uint256 _amount, bytes32[] calldata
    _merkleProof) external returns(uint256 _claimResult) {
100     require(_verify(msg.sender, _category, _amount, _merkleProof), "YayVesting:
    Invalid proof or wrong data");
101     require(CategoryNames(_category) != CategoryNames.EMPTY, "YayVesting:
    Invalid category");
102     require(_amount > 0, "YayVesting: Invalid amount");
103     require(block.timestamp >= tgeTimestamp, "YayVesting: TGE has not started
    yet");
        ...
143     alreadyRewarded[msg.sender] =
    alreadyRewarded[msg.sender].add(resultReward);
144     IERC20(token).transfer(msg.sender, resultReward);
145
```

```
146     emit Claim(msg.sender, _category, _amount, _merkleProof, resultReward,
    block.timestamp);
147
148     return(resultReward);
149 }
```

However, in this case, there is no impact from using the `transfer()` function since the `token` is the $YAY which is implemented with the handling of insufficient amount transfer from the `SafeMath` library.

5.6.2. Remediation

Inspex suggests replacing the `transfer()` functions of the tokens with `safeTransfer()` from OpenZeppelin's `SafeERC20` contract, for example:

YayVesting.sol

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity 0.6.12;
4  pragma experimental ABIEncoderV2;
5
6  import "@openzeppelin/contracts/math/SafeMath.sol";
7  import "@openzeppelin/contracts/cryptography/MerkleProof.sol";
8  import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
9  import "@openzeppelin/contracts/access/Ownable.sol";
10 import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
11
12 contract YayVesting is Ownable {
13     using SafeMath for uint256;
14     using SafeERC20 for IERC20;
```

YayVesting.sol

```
99  function claim(uint256 _category, uint256 _amount, bytes32[] calldata
    _merkleProof) external returns(uint256 _claimResult) {
100     require(_verify(msg.sender, _category, _amount, _merkleProof), "YayVesting:
    Invalid proof or wrong data");
101     require(CategoryNames(_category) != CategoryNames.EMPTY, "YayVesting:
    Invalid category");
102     require(_amount > 0, "YayVesting: Invalid amount");
103     require(block.timestamp >= tgeTimestamp, "YayVesting: TGE has not started
    yet");
        ...
143     alreadyRewarded[msg.sender] =
    alreadyRewarded[msg.sender].add(resultReward);
144     IERC20(token).safeTransfer(msg.sender, resultReward);
145
146     emit Claim(msg.sender, _category, _amount, _merkleProof, resultReward,
```

```
        block.timestamp);
147
148         return(resultReward);
149     }
```

5.7. Improper Function Visibility

ID	IDX-007
Target	BEP20YAY
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Yay Games team has changed the name of `burnBlackFunds()` function to `takeBlackFunds()` and resolved this issue as suggested in commit `1499d04af27b92ddaad39def4ab6e272708d947c`.

5.7.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `burnBlackFunds()` function of the `BEP20YAY` contract is set to public and it is never called from any internal function.

BEP20YAY.sol

```
645 function burnBlackFunds(address target, uint256 amount) public onlyOwner {
646     require(isBlacklisted(target));
647     _burn(target, amount);
648 }
```

5.7.2. Remediation

Inspex suggests changing the `burnBlackFunds()` function visibility to external if they are not called from any internal function as shown in the following example:

BEP20YAY.sol

```
645 function burnBlackFunds(address target, uint256 amount) external onlyOwner {
646     require(isBlacklisted(target));
647     _burn(target, amount);
648 }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] “OWASP Risk Rating Methodology.” [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE