TECHFUND Audit Score

Smart contract Audit Report

BOBG Audit

Completed on 2022-10-12

Score **POSITIVE**

Risk level		
	Critical	0
	High	0
	Medium	1
	Low	1
	Note	4

Risk level detail

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

The tester arrives at the likelihood and impact estimates, they can now combine them to get a final severity rating for this risk. Note that if they have good business impact information, they should use that instead of the technical impact information.

https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

Vulnerability Review

Number of warnings

Compiler Version	1
State Variable Default Visibility	0
Data Bounding / sanity checks	1
In-Correct Calculation Sequence	1
Integer Overflow / Underflow	0
Parity Multisig Bug	0
Callstack Depth Attack	0
Expensive Loop	1
Unwanted Boolean comparison	1
Re-Entrancy	1
Double Withdrawal	0

TECHFUND Audit Score

Smart contract Audit Report

Compiler Version

1

Solidity version used is inconsistent & not fixed. It is highly recommended to update the Solidity version to the latest. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statements and using a fixed version of Solidity upon understanding the version specifications.

```
pragma solidity ^0.8.0;
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import "./TokenPaymentSplitter.sol";
contract DepositContract is Initializable, OwnableUpgradeable, TokenPaymentSplitter {
    using SafeERC20Upgradeable for IERC20Upgradeable;
```

Data Bounding / sanity checks

1

Data unity check is not properly done for _paymentToken variable. For all address inputs it is highly recommended to check at least “zero check” to make sure that address is not equal to zero.

```
function initialize(address[] memory _payees, uint256[] memory _shares,
address _paymentToken, uint256 _releaseTime, uint256 _vestingMonth) public initializer {
    __Ownable_init();
    require(_releaseTime > block.timestamp);
    require(
        _payees.length == _shares.length,
        "TokenPaymentSplitter: payees and shares length mismatch"
    );
    require(_payees.length > 0, "TokenPaymentSplitter: no payees");
    for (uint256 i = 0; i < _payees.length; i++) {
        _addPayee(_payees[i], _shares[i]);
    }
    paymentToken = _paymentToken;
    releaseTime = _releaseTime;
    vestingMonth = _vestingMonth;
}
```

In-Correct Calculation Sequence

1

Solidity has bad support for precision and as a result when performing multiplication before division it can lead to loss of precision. For example if we do (a / b) * c and b > a , the result will be zero but if we use (a * c) / n the result will be correct.

In **claimableAmount** function the pattern is not correctly followed and it may lead to wrong results. It is highly suggested to follow “divide before any multiply” rule to avoid any issue, otherwise it can potentially return incorrect “claimable” value to user.

```
function claimableAmount() public view returns (uint256) {
    address account = msg.sender;
    require(block.timestamp >= releaseTime, "Still in lock-up period"); //ロックアップ期間中に引き出すことはできない
    uint256 tokenTotalReceived = IERC20Upgradeable(paymentToken).balanceOf(address(this)) + _totalTokenReleased; //Deposit Contractに送られた総トークン量
    uint256 totalAsset = (tokenTotalReceived * _shares[account]) / _totalShares; //アドレスに割り当てられた総トークン量
    uint256 paymentDaily = totalAsset / (365 * vestingMonth / 12); //一日あたりのリリース量
    uint256 vestingTotalDays = 365 * vestingMonth / 12; //ベスティングの総日数
    uint256 vestingDaysElapsed = (block.timestamp - releaseTime) / 60 / 60 / 24; //ベスティングの経過日数
    uint256 claimable; //claim可能なトークン量
    if (vestingDaysElapsed > vestingTotalDays) {
        claimable =  totalAsset - _tokenReleased[account]; //ベスティング期間が終わった後にはclaimable amountが増加しない
    } else {
        claimable = paymentDaily * vestingDaysElapsed - _tokenReleased[account];
    }
    return claimable;
}
```

(a/b) * c - d

Expensive Loop

1

Although we didn't find any major issue because of it, but it is suggested **not** to have potentially costly operations in a loop. It is always suggested to utilise "from - to" pattern with checks for maximum loop length. But since there is no critical issue with current implementation it is shown as "Note" level issue.

```
function initialize(address[] memory _payees, uint256[] memory _shares,
address _paymentToken, uint256 _releaseTime, uint256 _vestingMonth) public initializer {
    __Ownable_init();
    require(_releaseTime > block.timestamp);
    require(
        _payees.length == _shares.length,
        "TokenPaymentSplitter: payees and shares length mismat
    );
    require(_payees.length > 0, "TokenPaymentSplitter: no paye
    for (uint256 i = 0; i < _payees.length; i++) {
        _addPayee(_payees[i], _shares[i]);
    }
    paymentToken = _paymentToken;
    releaseTime = _releaseTime;
    vestingMonth = _vestingMonth;
}
```

```
// 初期化の際に使用する（引き出し可能なアドレスと分配率を指定する）
function _addPayee(address account, uint256 shares_) internal {
    require(
        account != address(0),
        "TokenPaymentSplitter: account is the zero address"
    );
    require(shares_ > 0, "TokenPaymentSplitter: shares are 0");
    require(
        _shares[account] == 0,
        "TokenPaymentSplitter: account already has shares"
    );
    uint entityIndex = _payees.length;
    _payees.push(account);
    _shares[account] = shares_;
    _totalShares = _totalShares + shares_;
    accountExists[account] = true;
    addressesEntityIndex[account] = entityIndex;
    emit PayeeAdded(account, shares_);
}
```

TECHFUND Audit Score

Smart contract Audit Report

Unwanted Boolean comparison

1

This is from code-sanity point of view and do not have any impact on security. It is suggested to use boolean constants for comparison directly as do not need to be compare to `true` or `false`.

```
function changeAddress(address oldAccount, address newAccount) external onlyOwner {
    require(accountExists[oldAccount] == true, "Account does not exist");
    require(accountExists[newAccount] == false, "Account already exists"); //既に登録済みのアドレスを指定できない（以下の処理が無効となってしまうため）
    uint addressEntityIndex = addressesEntityIndex[oldAccount];
    // アップデート処理
    _payees[addressEntityIndex] = newAccount;
    _shares[newAccount] = _shares[oldAccount];
```

Re-Entry

1

A reentrancy attack can occur when you create a function that makes an external call to another untrusted contract before it resolves any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the effects were resolved.

In **TokenPaymentSplitter.claim** function emits an event after an external call. However, which means we can get control at a point when event has not been fired. Which opens up the possibility of re-entry. Please follow “Check => Effect => Interaction” pattern to prevent such issues.

```
    }
    require(claimable != 0, "TokenPaymentSplitter: account is not due payment"); //claim可能なトークン量が現時点でゼロ
    require(claimable >= amount, "More than claimable amount"); //指定amountが現在claim可能なトークン量を超えている
    _tokenReleased[account] = _tokenReleased[account] + amount; //claim済みトークンを更新
    _totalTokenReleased = _totalTokenReleased + amount; //Deposit Contract全体のclaim済みトークンを更新
    IERC20Upgradeable(paymentToken).safeTransfer(account, amount); //指定amountを指定アドレスに送金
    emit PaymentReleased(account, amount); Event emitted after external call
}
```

External Call