# SMART CONTRACT AUDIT REPORT

for

# YYDS

Prepared By: Xiaomi Huang

PeckShield

November 8, 2023

## Document Properties

| | |
|---|---|
| Client | YYDS |
| Title | Smart Contract Audit Report |
| Target | YYDS |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Jinzhuo Shen, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | November 8, 2023 | Xuxian Jiang | Final Release |
| 1.0-rc | November 7, 2023 | Xuxian Jiang | Release Candidate #1 |

## Contact

For more information about this document and its contents, please contact PeckShield Inc.



| | |
|---|---|
| Name | Xiaomi Huang |
| Phone | +86 183 5897 7782 |
| Email | contact@peckshield.com |

# Contents

# 1 Introduction

Given the opportunity to review the design document and related source code of the YYDS token contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract can be further improved due to the presence of certain issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

## 1.1 About YYDS

YYDS is a deflationary token launched on Binance Smart Chain. The total amount of tokens is fixed at 1 million, of which 600,000 are used to add liquidity and 400,000 are used for ecological construction. There is no pre-sale for the issuance of tokens. The token serves as the governance token of the YYDS community and is also the equity certificate of the YYDS ecosystem. YYSD automatically deflates every 24 hours, and the currency price automatically rises to encourage community development. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of YYDS

| Item | Description |
|---:|---|
| Name | YYDS |
| Type | ERC20 Token Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | November 8, 2023 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/yydsfinance/yyds.git (2148dce)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/yydsfinance/yyds.git (287d262)

## 1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.2: Vulnerability Severity Classification

| Impact \ Likelihood | *High* | *Medium* | *Low* |
|---|---|---|---|
| *High* | Critical | High | Medium |
| *Medium* | High | Medium | Low |
| *Low* | Medium | Low | Low |

We perform the audit according to the following procedures:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.3: The Full List of Check Items

| Category | Check Item |
|---|---|
| **Basic Coding Bugs** | Constructor Mismatch |
| | Ownership Takeover |
| | Redundant Fallback Function |
| | Overflows & Underflows |
| | Reentrancy |
| | Money-Giving Bug |
| | Blackhole |
| | Unauthorized Self-Destruct |
| | Revert DoS |
| | Unchecked External Call |
| | Gasless Send |
| | Send Instead of Transfer |
| | Costly Loop |
| | (Unsafe) Use of Untrusted Libraries |
| | (Unsafe) Use of Predictable Variables |
| | Transaction Ordering Dependence |
| | Deprecated Uses |
| | Approve / TransferFrom Race Condition |
| **ERC20 Compliance Checks** | Compliance Checks (Section 3) |
| **Additional Recommendations** | Avoiding Use of Variadic Byte Array |
| | Using Fixed Compiler Version |
| | Making Visibility Level Explicit |
| | Making Type Inference Explicit |
| | Adhering To Function Declaration Strictly |
| | Following Other Best Practices |

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

# 2 Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the `YYDS` token contract. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 0 | |
| Medium | 1 | |
| Low | 3 | |
| Informational | 0 | |
| Total | 4 | |

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

## 2.2 Key Findings

Overall, no ERC20 compliance issue was found and our detailed checklist can be found in Section 3. While there is no critical or high severity issue, the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key YYDS Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Interval Restriction Bypass in Token Buys | Business Logic | Confirmed |
| PVE-002 | Low | Accommodation of ERC20 Noncompliant Tokens | Coding Practices | Resolved |
| PVE-003 | Low | Improved Parameter Validation Upon Their Changes | Coding Practices | Resolved |
| PVE-004 | Medium | Trust Issue Of Admin Keys | Security Features | Confirmed |

Besides recommending specific countermeasures to mitigate the above issue(s), we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks and Section 4 for elaboration of reported issues.

# 3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

| Item | Description | Status |
|---|---|---|
| **name()** | Is declared as a public view function | ✓ |
| | Returns a string, for example "Tether USD" | ✓ |
| **symbol()** | Is declared as a public view function | ✓ |
| | Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length | ✓ |
| **decimals()** | Is declared as a public view function | ✓ |
| | Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required | ✓ |
| **totalSupply()** | Is declared as a public view function | ✓ |
| | Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment | ✓ |
| **balanceOf()** | Is declared as a public view function | ✓ |
| | Anyone can query any address' balance, as all data on the blockchain is public | ✓ |
| **allowance()** | Is declared as a public view function | ✓ |
| | Returns the amount which the spender is still allowed to withdraw from the owner | ✓ |

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited `YYDS` token contract. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

| Item | Description | Status |
|---|---|---|
| **transfer()** | Is declared as a public function | ✓ |
| | Returns a boolean value which accurately reflects the token transfer status | ✓ |
| | Reverts if the caller does not have enough tokens to spend | ✓ |
| | Allows zero amount transfers | ✓ |
| | Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers) | ✓ |
| | Reverts while transferring to zero address | ✓ |
| **transferFrom()** | Is declared as a public function | ✓ |
| | Returns a boolean value which accurately reflects the token transfer status | ✓ |
| | Reverts if the spender does not have enough token allowances to spend | ✓ |
| | Updates the spender's token allowances when tokens are transferred successfully | ✓ |
| | Reverts if the from address does not have enough tokens to spend | ✓ |
| | Allows zero amount transfers | ✓ |
| | Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers) | ✓ |
| | Reverts while transferring from zero address | ✓ |
| | Reverts while transferring to zero address | ✓ |
| **approve()** | Is declared as a public function | ✓ |
| | Returns a boolean value which accurately reflects the token approval status | ✓ |
| | Emits Approval() event when tokens are approved successfully | ✓ |
| | Reverts while approving to zero address | ✓ |
| **Transfer()** event | Is emitted when tokens are transferred, including zero value transfers | ✓ |
| | Is emitted with the from address set to $address(0x0)$ when new tokens are generated | ✓ |
| **Approval()** event | Is emitted on any successful call to approve() | ✓ |

In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements, but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

| Feature | Description | Opt-in |
|---|---|---|
| **Deflationary** | Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls | ✓ |
| **Rebasing** | The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address | — |
| **Pausable** | The token contract allows the owner or privileged users to pause the token transfers and other operations | — |
| **Whitelistable** | The token contract allows the owner or privileged users to whitelist a specific address such that only token transfers and other operations related to that address are allowed | ✓ |
| **Mintable** | The token contract allows the owner or privileged users to mint tokens to a specific address | — |
| **Burnable** | The token contract allows the owner or privileged users to burn tokens of a specific address | — |

# 4 | Detailed Results

## 4.1 Interval Restriction Bypass in Token Buys

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: YYDS
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

The YYDS token contract has a built-in rule in enforcing that a given EOA wallet can only buy token once within 40 seconds. Our analysis on this rule indicates that it may be bypassed.

To elaborate, we show below the related _tokenTransfer() routine, which will be invoked for every token transfer. Our analysis shows that the above rule is enforced (for the token buy) as follows: `require(block.timestamp - lastTimeTx[tx.origin] >= buyInterval)`. With that, it is possible to launch a Sybil attack to always initiate the buy from a fresh EOA wallet and then receive the tokens in the actual user wallet.

```
    function _tokenTransfer(address sender, address recipient, uint256 amount, bool takeFee) private {
        if(takeFee) {
            require(startTime > 0, "not time");
            uint feeToThis;
            if(isPair[sender]) { //buy
                if (limitedTrade) {
                    require(block.timestamp - lastTimeTx[tx.origin] >= buyInterval, "trading after a while");
                    lastTimeTx[tx.origin] = block.timestamp;
                    address[] memory path = new address[](2);
                    path[0] = USDT;
                    path[1] = address(this);
                    uint[] memory amountsIn = IUniswapV2Router02(ROUTER).getAmountsIn(amount, path);
                    require(amountsIn[0] <= maxTokenVaulePerTx, "max usdt limit");
```

```
                }
                feeToThis = buyMarketingFee;
            } ...
        }
        super._transfer(sender, recipient, amount);
    }
```

Listing 4.1: `YYDS::_tokenTransfer()`

**Recommendation** Revisit the above rule enforcement to ensure the limit is properly honored.

**Status** The issue has been resolved as the enforcement is validated on `tx.origin`, not `msg.sender`.

## 4.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `YYDS`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

### Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
```

```
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 4.2: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()` , In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 4.3: `SafeERC20::safeApprove()`

In current implementation, if we examine the `TokenReceiver::constructor()` routine that is designed to approve the deployer the full allowance in spending the given tokens. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (line 476).

```
474 contract TokenReceiver {
475     constructor(address token) {
476         IERC20(token).approve(msg.sender, type(uint).max);
477     }
478 }
```

Listing 4.4: `TokenReceiver::constructor()`

Note the `swapAndDividend()` routine in the same contract can be similarly improved by making use of `safeTransferFrom()`.

**Recommendation** Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

**Status** The issue has been fixed by this commit: `9239dc8`.

## 4.3 Improved Parameter Validation Upon Their Changes

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `YYDS`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `YYDS` token is no exception. Specifically, if we examine the `YYDS` contract, it has defined a number of protocol-wide risk parameters, such as `buyMarketingFee` and `sellMarketingFee`. In the following, we show the corresponding routines that allow for their changes.

```
    function setBuyMarketingFee(uint256 _buyMarketingFee) external onlyOwner {
        buyMarketingFee = _buyMarketingFee;
    }

    function setMarketingWallet(address _marketingAddr) external onlyOwner {
        marketingAddr = _marketingAddr;
    }

    function setShareholderAddr(address _shareholderAddr) external onlyOwner {
        shareholderAddr = _shareholderAddr;
    }

    function setTreasureAddr(address _treasureAddr) external onlyOwner {
        treasureAddr = _treasureAddr;
    }

    function setSellMarketingFee(uint256 _sellMarketingFee) external onlyOwner {
        sellMarketingFee = _sellMarketingFee;
    }

    function setNumTokensSellToSwap(uint256 value) external onlyOwner {
        numTokenValueSellToSwap = value;
```

```
    }
```

Listing 4.5: Example Protocol Parameter Updates in YYDS

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `buyMarketingFee` may charge unreasonably high fee in its buy, hence incurring cost to borrowers or hurting the token adoption.

**Recommendation** Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

**Status** The issue has been fixed by this commit: `9239dc8`.

## 4.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `YYDS`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

### Description

In the `YYDS` token contract, there is a privileged admin account `owner` that plays a critical role in regulating the token-wide operations (e.g., configure fees and blacklist accounts). In the following, we show the representative function potentially affected by this privilege.

```
    function setVault(address _vault) external onlyOwner {
        VAULT = _vault;
    }

    function setLimitedTrade(bool _limitedTrade) external onlyOwner {
        limitedTrade = _limitedTrade;
    }

    function setBuyInterval(uint _buyInterval) external onlyOwner {
        buyInterval = _buyInterval;
    }

    function setPair(address pair, bool value) external onlyOwner {
        isPair[pair] = value;
    }
```

```
    function excludeFromFees(address account, bool excluded) public onlyOwner {
        isExcludedFromFees[account] = excluded;
    }

    function excludeMultipleAccountsFromFees(address[] calldata accounts, bool excluded)
         public onlyOwner {
        for(uint256 i = 0; i < accounts.length; i++) {
            isExcludedFromFees[accounts[i]] = excluded;
        }
    }

    function setMaxTokenVaulePerTx(uint _maxTokenVaulePerTx) external onlyOwner {
        maxTokenVaulePerTx = _maxTokenVaulePerTx;
    }
```

Listing 4.6: An Example Privileged Operation in YYDS

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the new owner to modify a number of sensitive system parameters, which may directly undermine the assumption of the token design.

**Recommendation** Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

**Status** The issue has been confirmed by the team.

# 5 Conclusion

In this security audit, we have examined the YYDS token design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, although no critical level vulnerabilities were discovered, we identified several issues that need to be promptly addressed. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

# References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[8] PeckShield. PeckShield Inc. https://www.peckshield.com.