SMART CONTRACT AUDIT REPORT

Yala Staking

OCTOBER 2025

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **Yala Staking** to review smart contract implementation. The assessment was conducted in accordance with our systematic approach to evaluate potential security issues based upon customer requirement. The report provides detailed recommendations to resolve the issue and provide additional suggestions or recommendations for improvement.

The outcome of the assessment outlined in chapter 3 provides the system's owners a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendations that will resolve the underlying technical issue.

1.1 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10] which is the gold standard in risk assessment using the following risk models:

- **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**: determine the overall criticality of the risk.

Likelihood can be: High, Medium and Low and impact are categorized into: High, Medium, Low, Informational. Severity is determined by likelihood and impact and can be classified into five categories accordingly: Critical, High, Medium, Low, Informational shown in table 1.1.

Likelihood \ IMPACT	Informational	Low	Medium	High
High	INFO	MEDIUM	HIGH	CRITICAL
Medium	INFO	LOW	MEDIUM	HIGH
Low	INFO	LOW	LOW	MEDIUM

Table 1.1 Overall Risk Severity

To evaluate the risk, we will be going through a list of items, and each would be labelled with a severity category. The audit was performed with a systematic approach guided by a comprehensive assessment list carefully designed to identify known and impactful security issues. If our tool or analysis does not identify any issue, the contract can be considered safe regarding the assessed item. For any discovered issue, we might further deploy contracts on our private test environment 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 1.2.

- **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.
- **Code and business security testing**: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- **Additional Recommendations**: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Category	**Assessment Item**
Basic Coding Assessment	• Apply Verification Control • Authorization Access Control • Forged Transfer Vulnerability • Forged Transfer Notification • Numeric Overflow • Transaction Rollback Attack • Transaction Block Stuffing Attack • Soft Fail Attack • Hard Fail Attack • Abnormal Memo • Abnormal Resource Consumption • Secure Random Number

Advanced Source Code Scrutiny	• Asset Security • Cryptography Security • Business Logic Review • Source Code Functional Verification • Account Authorization Control • Sensitive Information Disclosure • Circuit Breaker • Blacklist Control • System API Call Analysis • Contract Deployment Consistency Check • Abnormal Resource Consumption
Additional Recommendations	• Semantic Consistency Checks • Following Other Best Practices

Table 1.2: The Full List of Assessment Items

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development.

2. FINDINGS OVERVIEW

2.1 Project Info And Contract Address

Project Name	Audit Time	Language
Yala Staking	16/10/2025 - 17/10/2025	Solidity

Repository
https://github.com/yalaorg/yala-staking

Commit Hash
100aa57d331418857dbf82d05427c07cbc91485a

2.2 Summary

Severity	Found
CRITICAL	0
HIGH	0
MEDIUM	1
LOW	2
INFO	3

2.3 Key Findings

Severity	Findings Title	Status
MEDIUM	Deposit Cap Can Be Set Below Current Deposit Fill	Fixed
LOW	Lack of verification with current time	Fixed
LOW	Risk of overlap between beginTime and endTime	Fixed
INFO	Lack of zero address checking	Fixed
INFO	Missing event record	Fixed
INFO	constructor2 initialization can be called repeatedly	Fixed

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Deposit Cap Can Be Set Below Current Deposit Fill

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/core/VaultController.sol

DESCRIPTION:

In the YalaStaking contract, deposit and mint will verify whether depositFill exceeds depositCap, and then the administrator will modify the upper limit in _setDepositCap. Here, it may happen that the modified depositCap is less than the deposited depositFill.

```
// contracts/core/VaultController.sol
function _setDepositCap(uint256 _depositCap) internal {
    depositCap = _depositCap;
}
```

IMPACT:

If the modified depositCap is smaller than depositFill, the deposit and mint operations will fail.

RECOMMENDATIONS:

Add _depositCap validation to ensure the new deposit cap is greater than or equal to the current deposit fill:

```
function _setDepositCap(uint256 _depositCap) internal {
+   if (_depositCap < depositFill) {
+       revert InvalidDepositCap(_depositCap, depositFill);
+   }
    depositCap = _depositCap;
}
```

3.2 Lack of verification with current time

SEVERITY: LOW **STATUS:** Fixed

PATH:

contracts/core/VaultController.sol

DESCRIPTION:

The _setTime function of the VaultController contract will limit beginTime < endTime < withdrawable-Time when it is executed, but there is no comparison with the current time, so endTime may be earlier than the current time.

```
// contracts/core/VaultController.sol
function _setTime(uint256 beginTime, uint256 endTime, uint256
    withdrawableTime) internal {
     if (endTime < beginTime) {
         revert InvalidDepositTimeRange(beginTime, endTime);
     }
     if (withdrawableTime < endTime) {
         revert InvalidWithdrawableTime(withdrawableTime);
     }
     depositBeginTime = beginTime;
     depositEndTime = endTime;
     withdrawableBeginTime = withdrawableTime;
}
```

IMPACT:

If the endTime is earlier than the current time, the user will not be able to interact with the project normally.

RECOMMENDATIONS:

Make sure beginTime is after the current time. Ensure block.timestamp <= beginTime < endTime <= withdrawableTime:

```
function _setTime(uint256 beginTime, uint256 endTime, uint256
    withdrawableTime) internal {
+    if (beginTime < block.timestamp) {
+        revert InvalidBeginTime(block.timestamp, beginTime);
+    }
     if (endTime < beginTime) {
         revert InvalidDepositTimeRange(beginTime, endTime);
     }
     if (withdrawableTime < endTime) {
         revert InvalidWithdrawableTime(withdrawableTime);
     }
     depositBeginTime = beginTime;
     depositEndTime = endTime;
     withdrawableBeginTime = withdrawableTime;
}
```

3.3 Risk of overlap between beginTime and endTime

SEVERITY: LOW **STATUS:** Fixed

PATH:

contracts/core/VaultController.sol

DESCRIPTION:

The _setTime function of the VaultController contract will limit beginTime < endTime when executed, but ignore the equal case, so that the start time and end time can coincide.

IMPACT:

If the start and end times coincide, users will not be able to trade normally.

RECOMMENDATIONS:

Add boundary judgment to ensure beginTime is strictly less than endTime:

```
function _setTime(uint256 beginTime, uint256 endTime, uint256
    withdrawableTime) internal {
-    if (endTime < beginTime) {
+    if (endTime <= beginTime) {
         revert InvalidDepositTimeRange(beginTime, endTime);
     }
     if (withdrawableTime < endTime) {
         revert InvalidWithdrawableTime(withdrawableTime);
     }
     depositBeginTime = beginTime;
     depositEndTime = endTime;
     withdrawableBeginTime = withdrawableTime;
}
```

3.4 Lack of zero address checking

SEVERITY: INFO **STATUS:** Fixed

PATH:

`contracts/core/Factory.sol`

DESCRIPTION:

In the Factory contract, the constructor and setImplementation should perform zero address verification on the incoming address. The probability of this happening is very small, but it can make the code more standardized.

IMPACT:

While unlikely, passing a zero address could lead to contract deployment issues or unexpected behavior during implementation updates.

RECOMMENDATIONS:

Checks whether the passed address is equal to address(0) and throws a corresponding error. Add zero address validation to ensure defensive programming practices:

```
+ if (implementationAddress == address(0)) {
+     revert InvalidAddress();
+ }
```

3.5 Missing event record

SEVERITY: INFO **STATUS:** Fixed

PATH:

`contracts/core/Factory.sol`

DESCRIPTION:

The setImplementation function of the Factory contract will directly modify the value of the stakingImplementation variable. It is recommended to issue an event for recording after execution.

IMPACT:

Without event emission, off-chain systems and users cannot easily track when the implementation address changes, reducing transparency and making debugging more difficult.

RECOMMENDATIONS:

Add the Event modified by stakingImplementation in the setImplementation function. Emit an event after modifying the implementation address to provide transparency and enable off-chain monitoring:

```
+ event ImplementationUpdated(address indexed oldImplementation, address
    indexed newImplementation);

function setImplementation(address newImplementation) external onlyOwner {
+    address oldImplementation = stakingImplementation;
     stakingImplementation = newImplementation;
+    emit ImplementationUpdated(oldImplementation, newImplementation);
}
```

3.6 constructor2 initialization can be called repeatedly

SEVERITY: INFO **STATUS:** Fixed

PATH:

contracts/core/YalaStaking.sol

DESCRIPTION:

The constructor2 function in the YalaStaking contract is used to initialize the cloned staking contract. When the same parameters are used in deployNewStaking, cloneDeterministic will revert if a contract with the same address already exists, preventing repeated initialization. Adding an initialization check inside constructor2 further standardizes the code and ensures defensive programming practices.

IMPACT:

While the factory pattern provides some protection, lacking an internal initialization guard could potentially allow reinitialization in edge cases or if the function is called through other means.

RECOMMENDATIONS:

Add an initialization flag to prevent repeated initialization and ensure defensive programming:

```
+ bool private _initialized;

function constructor2(address admin, string memory name_, string memory
   symbol_) external {
    if (msg.sender != factory) {
        revert OnlyFactory();
    }
+   require(!_initialized, "Already initialized");
+   _initialized = true;
    _transferOwnership(admin);
    _name = name_;
    _symbol = symbol_;
}
```

4. CONCLUSION

In this audit, we thoroughly analyzed **Yala Staking** smart contract implementation. The problems found are described and explained in detail in Section 3. The problems found in the audit have been communicated to the project leader. We therefore consider the audit result to be **PASSED**.

To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5. APPENDIX

5.1 Basic Coding Assessment

5.1.1 Apply Verification Control

Description	The security of apply verification
Result	Not found
Severity	CRITICAL

5.1.2 Authorization Access Control

Description	Permission checks for external integral functions
Result	Not found
Severity	CRITICAL

5.1.3 Forged Transfer Vulnerability

Description	Assess whether there is a forged transfer notification vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.4 Transaction Rollback Attack

Description	Assess whether there is transaction rollback attack vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.5 Transaction Block Stuffing Attack

Description	Assess whether there is transaction blocking attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.6 Soft Fail Attack Assessment

Description	Assess whether there is soft fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.7 Hard Fail Attack Assessment

Description	Examine for hard fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.8 Abnormal Memo Assessment

Description	Assess whether there is abnormal memo vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.9 Abnormal Resource Consumption

Description	Examine whether abnormal resource consumption in contract processing
Result	Not found
Severity	CRITICAL

5.1.10 Random Number Security

Description	Examine whether the code uses insecure random number
Result	Not found
Severity	CRITICAL

5.2 Advanced Code Scrutiny

5.2.1 Cryptography Security

Description	Examine for weakness in cryptograph implementation
Result	Not found
Severity	HIGH

5.2.2 Account Permission Control

Description	Examine permission control issue in the contract
Result	Not found
Severity	MEDIUM

5.2.3 Malicious Code Behavior

Description	Examine whether sensitive behavior present in the code
Result	Not found
Severity	MEDIUM

5.2.4 Sensitive Information Disclosure

Description	Examine whether sensitive information disclosure issue present in the code
Result	Not found
Severity	MEDIUM

5.2.5 System API

Description	Examine whether system API application issue present in the code
Result	Not found
Severity	LOW

6. DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without ExVul's prior written consent.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts ExVul to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. ExVul's position is that each company and individual are responsible for their own due diligence and continuous security. ExVul's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

7. REFERENCES

[1] MITRE. CWE-191: Integer Underflow (Wrap or Wraparound). https://cwe.mitre.org/data/definitions/191.html.

[2] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html.

[3] MITRE. CWE-400: Uncontrolled Resource Consumption. https://cwe.mitre.org/data/definitions/400.html.

[4] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440.html.

[5] MITRE. CWE-684: Protection Mechanism Failure. https://cwe.mitre.org/data/definitions/693.html.

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

[7] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE CATEGORY: Resource Management Errors. https://cwe.mitre.org/data/definitions/399.html.

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

8. About Exvul Security

Premier Security for the Web3 Ecosystem

ExVul is a premier Web3 security firm committed to forging a secure and trustworthy decentralized ecosystem. Our elite team consists of security veterans from world-leading technology and blockchain security firms, including Huawei, YBB Captical, Qihoo 360, Amber, ByteDance, MoveBit, and PeckShield. Team member Nolan is ranked as a top-40 whitehat on Immunefi and is the platform's sole All-Star in the APAC region.

Our expertise covers the full spectrum of Web3 security. We conduct **meticulous smart contract audits**, having fortified thousands of projects on chains like Evm, Solana, Aptos, Sui etc. Our **Blockchain Protocol Audits** secure the core infrastructure of L1/L2 by uncovering deep-seated vulnerabilities. We also offer **comprehensive wallet audits** to protect user assets and provide **proactive web3 pentest**, enabling partners to neutralize threats before they strike.

Trusted by industry leaders, ExVul is the security partner for **OKX, Bitget, Cobo, Infini, Stacks, Aptos, Sui, CoreDAO, Sei** etc.

Contact

Website
www.exvul.com

Email
contact@exvul.com

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec

ExVul