Competitive Security Assessment

Aki_V3

Dec 20th, 2023

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

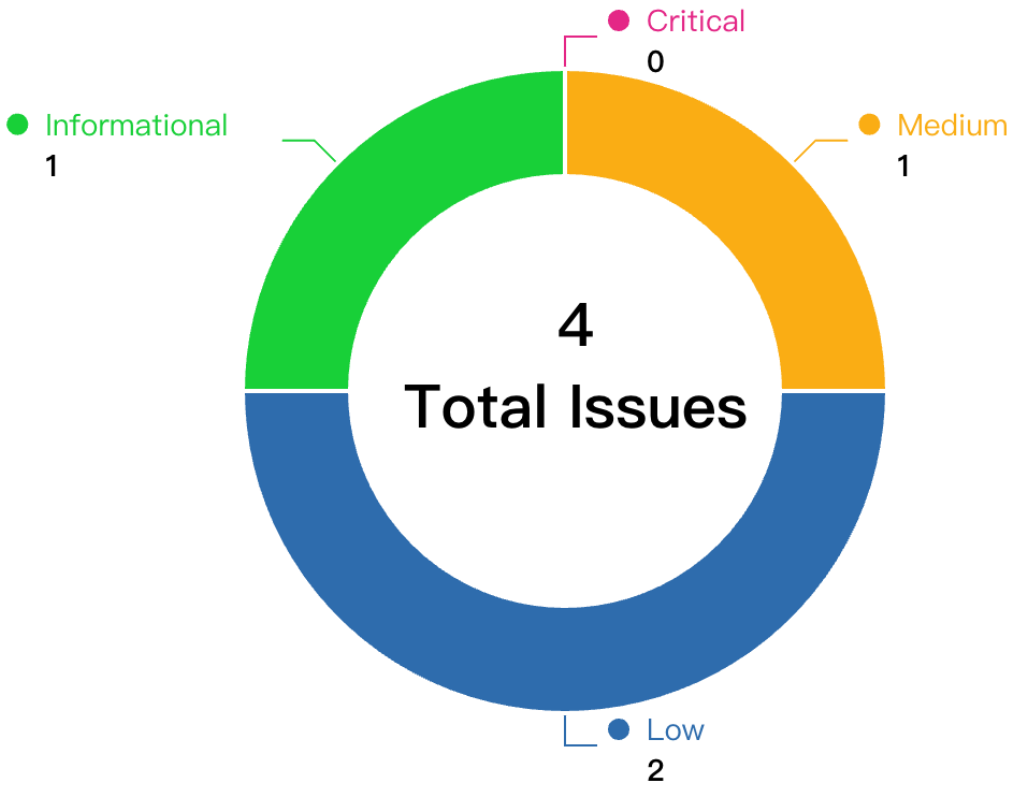
Project Detail

Project Name	Aki_V3
Platform & Language	Solidity
Codebase	• Code shared by zip file • audit file sha256 hash - 913ad21faa40bfcff9f38a805627c0939de209ad2cc39e229f0569b614f973f7 • final file sha256 hash - 4680f52b860993df8ce8c89f0ba163e115f92a65bcb81d607945233066224768
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
./AkiTokenDispenser3.sol	913ad21faa40bfcff9f38a805627c0939de209ad2cc39e229f0569b614f973f7

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
AK3-1	Unchecked ERC-20 `transfer()` and `transferFrom()` call	Code Style	Medium	Fixed	NoodleDonn 212, Secure3
AK3-2	User reward should not be set after reward claiming is started	Logical	Low	Fixed	infinityhack er, Secure3
AK3-3	`setShares()` lacks zero address validation	Logical	Low	Fixed	Secure3
AK3-4	`uint256 _shares` is always non-negative	Language Specific	Informational	Fixed	Secure3

AK3-1:Unchecked ERC-20 `transfer()` and `transferFrom()` call

Category	Severity	Client Response	Contributor
Code Style	Medium	Fixed	NoodleDonn212, Secure3

Code Reference

- code/AkiTokenDispenser3.sol#L67
- code/AkiTokenDispenser3.sol#L70
- code/AkiTokenDispenser3.sol#L80
- code/AkiTokenDispenser3.sol#L103
- code/AkiTokenDispenser3.sol#L134

```
67:function depositTokensIntoPool(uint256 _amount) external onlyOwner validRewardToken {

70:rewardToken.transferFrom(msg.sender, address(this), _amount);

80:function claimTokens() external validRewardToken {

103:rewardToken.transfer(msg.sender, claimAmount);

134:rewardToken.transfer(owner(), _amount);
```

Description

NoodleDonn212 : Summary:

The AkiTokenDispenser3 smart contract's depositTokensIntoPool and claimTokens functions do not adhere to best practices as outlined in the ERC20 standard regarding the handling of return values. This oversight can result in scenarios where these functions incorrectly assume success even if the underlying transfer or transferFrom calls fail.

Vulnerability Details:

The ERC20 standard specifies that the transfer and transferFrom functions should return a boolean value indicating the success or failure of the operation. However, the AkiTokenDispenser3 contract's depositTokensIntoPool and claimTokens functions interact with an ERC20 token contract without checking the return value of these calls. Specifically, the depositTokensIntoPool function calls rewardToken.transferFrom(msg.sender, address(this), _amount) and the claimTokens function calls rewardToken.transfer(msg.sender, claimAmount) without validating the returned boolean value.

Impact:

Failure to check the return value of the transfer and transferFrom calls can lead to misleading transaction outcomes where the contract functions proceed as if the token transfers were successful, even though the actual token transfers did

not occur. This can cause discrepancies in balance tracking, incorrect reward distribution, and potentially result in loss of funds or incorrect accounting.
Secure3 : The return value of `transfer` and `transferFrom` function is not checked, and it can be a failure.

Recommendation

NoodleDonn212 : Recommendations:
Modify the depositTokensIntoPool and claimTokens functions to check the return value of the transferFrom and transfer calls, respectively, and handle any potential failure appropriately. If the transferFrom or transfer calls return false, the contract functions should revert the transaction to ensure a consistent state and prevent false positives. Consider using OpenZeppelin's SafeERC20 library, which provides wrapper functions such as safeTransfer and safeTransferFrom. These functions are designed to ensure that token transfers are successful and revert the transaction if they fail, preventing silent transfer failures and unexpected behavior in the contract.
Secure3 : check the return value of the `transfer` and `transferFrom` to make sure the token transfer is successful, or simply use the `SafeERC20` - https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol lib

Client Response

Fixed. used safe transfers from openzeppelin

AK3-2:User reward should not be set after reward claiming is started

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	infinityhacker, Secure3

Code Reference

- code/AkiTokenDispenser3.sol#L42
- code/AkiTokenDispenser3.sol#L42-L53
- code/AkiTokenDispenser3.sol#L59
- code/AkiTokenDispenser3.sol#L95-L96

```
42:function setShares(address _address, uint256 _shares) public onlyOwner {

42:function setShares(address _address, uint256 _shares) public onlyOwner {
43:        require(_shares >= 0, "Shares must be non-negative");
44:
45:        uint256 previousShares = userRewardInfo[_address].shares;
46:
47:        // Deduct previous shares and add new shares
48:        totalShares = totalShares - previousShares + _shares;
49:
50:        // Increment the number of winners not claimed if the user has not claimed rewards before
51:        if (previousShares == 0) {
52:            winnersNotClaimed++;
53:        }

59:function batchSetShares(address[] calldata _addresses, uint256[] calldata _shares) external onlyO
wner {

95:userRewardInfo[msg.sender].hasClaimedReward = true;
96:        userRewardInfo[msg.sender].claimedRewardAmount = claimAmount;
```

Description

infinityhacker : In `claimTokens` function, after reward claiming is start, all eligible user can claim reward tokens in same ratio, it was calculated by

```
  (rewardPoolBalance * userShares) / totalShares;
```

if owner change user share at reward claiming period, the share / rewardToken ratio will be changed. which result in unexpected error.

Secure3 : the two `setShares()` and `claimTokens()` functions are related to each other.

When `_address` has previously claimed the reward, the `userRewardInfo[_address].shares` will be set to 0 and `userRewardInfo[_address].hasClaimedReward` to be true.

if Aki owner calls `setShares` again with a previously claimed address, the `previousShares` will be zero and `winnersNotClaimed++` will be increased. Depending on whether `winnersNotClaimed` represents the unique EOA wallet address or the total number of rewards claimed, the logic could be wrong.

In addition, `mapping(address => UserRewardInfo) public userRewardInfo` cannot handle multiple claims of the same EOA address because the `hasClaimedReward` is set to true and never set back to false when `setShares` is called.

in summary

```
        if (previousShares == 0) {
            winnersNotClaimed++;  //@audit -> this indicates that multiple claims is allowed
        }
```

Recommendation

infinityhacker : Add `claimingEnabled` check when set user share

Secure3 : if multiple claims are allowed, in the `setShares()`

```
        userRewardInfo[_address].shares = _shares;
+       userRewardInfo[_address].hasClaimedReward = false;
+       userRewardInfo[_address].claimedRewardAmount = 0 ; // depends if this is claim per round or the total claims combined, if latter in the claimTokens() L96 use `+=` instead
    }
```

Otherwise, add below check in the `setShares()` to prevent address can be set multiple times

```
+ require(!userRewardInfo[_addressr].hasClaimedReward, "Reward already claimed");
```

Client Response

Fixed. added checks to disallow setting shares after claiming

AK3-3: `setShares()` lacks zero address validation

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	Secure3

Code Reference

- code/AkiTokenDispenser3.sol#L42-L43

```
42:function setShares(address _address, uint256 _shares) public onlyOwner {
43:         require(_shares >= 0, "Shares must be non-negative");
```

Description

Secure3 : `_address` can be zero and if that's the case, the shares will be assigned to a zero address and the `winnersNotClaimed` will be inaccurately increase one.

Recommendation

Secure3 : check the `_address`

```
require(address(_address) != address(0), "_address must not be zero");
```

Client Response

Fixed. added address non-zero check when setting shares

AK3-4: `uint256 _shares` is always non-negative

Category	Severity	Client Response	Contributor
Language Specific	Informational	Fixed	Secure3

Code Reference

- code/AkiTokenDispenser3.sol#L42-L43

```
42:function setShares(address _address, uint256 _shares) public onlyOwner {
43:         require(_shares >= 0, "Shares must be non-negative");
```

Description

Secure3 : the check in `setShares()` function

```
  require(_shares >= 0, "Shares must be non-negative");
```

check is redundant as the `uint256 _shares` is unsigned integer and it is always >= 0

Recommendation

Secure3 : remove the redundant check

Client Response

Fixed. removed this line

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 Invoices, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3’s prior written consent in each instance.

This report is not an “endorsement” or “disapproval” of any particular project or team. This report is not an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3’s position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.