Competitive Security Assessment

Aki Protocol

Mar 8th, 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 Protocol
Platform & Language	Solidity
Codebase	• https://github.com/akiprotocol-dev/redEnvelopes • audit commit - 946e682e98a1ae704a929f2b2f42e92e46c5c5a3 • final commit - 46d478f2a76fa7851ac69be777525727759ccdbf
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

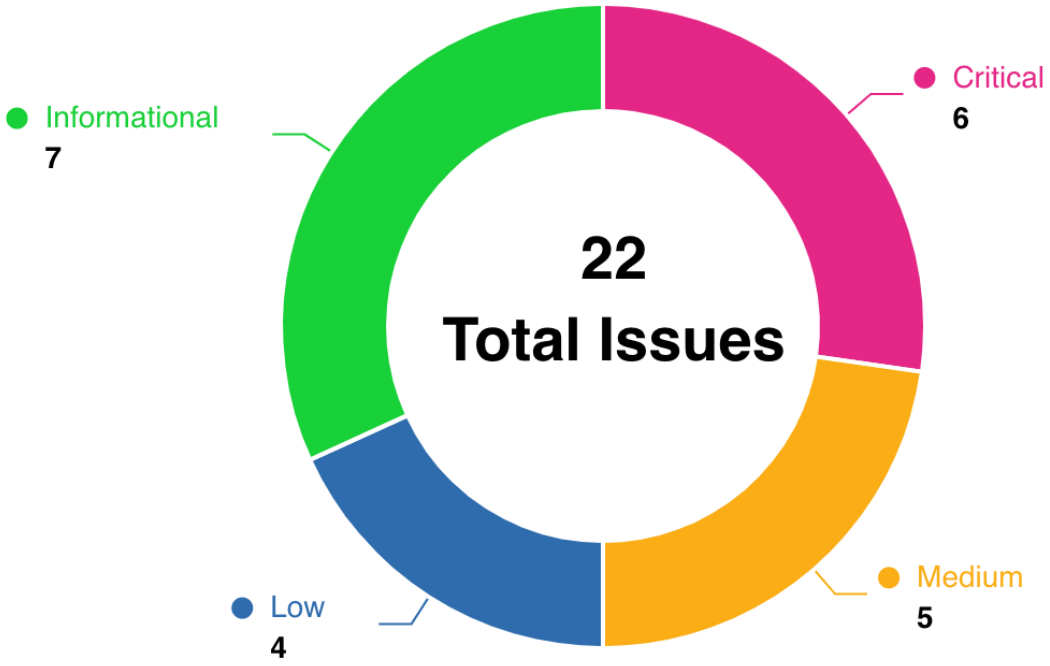
Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	6	0	1	5	0	0
Medium	5	0	1	3	1	0
Low	4	0	1	3	0	0
Informational	7	0	1	5	1	0

Audit Scope

File	Commit Hash
CampaignEnvelopeERC20.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
RedEnvelopeZkERC20ChainLink.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
AkiBadgeA1.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
RedEnvelopeMerkle.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
AkiBadge.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
RedEnvelopeMerkleERC20.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
INJDojoSignUp.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
SynTraders.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
NFTByLevel.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
AkiOracleMVP.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
SynPioneer.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
Aki×EthSign.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
RedEnvelopeERC20String.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
RedEnvelopeMerkleERC721.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
PullPaymentERC1155Envelope.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
IPullPaymentEnvelope.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
MerkelProofVerify.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
AkiRewardSplitter.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
RedEnvelopeMerkleERC1155.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
RedEnvelopeERC20.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
IEnvelope.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
BigBlueBunny.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
AkiTreasury.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
RedEnvelope.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
AkiBadgeB1.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3
PullPaymentERC20Envelopes.sol	946e682e98a1ae704a929f2b2f42e92e46c5c5a3

Code Assessment Findings



ID	Name	Category	Severity	Status	Contributor
AKI-1	Gas optimization for RedEnvelopeZkERC20ChainLink	Gas Optimization	Informational	Fixed	comcat
AKI-2	Envelopes should be marked to prevent multiple claims	Logical	Critical	Fixed	comcat, Kong7ych3
AKI-3	`PullPaymentERC1155Envelopes::openEnvelope` should transfer tokens out	Logical	Medium	Mitigated	comcat, Kong7ych3, zzzix
AKI-4	Lack of support for fee-on-transfer tokens	Logical	Low	Fixed	comcat

AKI-5	Use of weak pseudo-random number generator	Weak Sources of Randomness	Medium	Acknowledged	comcat, Kong7ych3, BradMoonUESTC
AKI-6	Lack of checks for user input parameters	Logical	Informational	Fixed	comcat, BradMoonUESTC
AKI-7	Multiple contracts are missing necessary events	Code Style	Informational	Mitigated	comcat
AKI-8	Critical parameters modification lacks permission control	Logical	Critical	Fixed	comcat, Xi_Zi, Kong7ych3
AKI-9	Miss access control for the `AkiRewardSpliter::addPayment`	Logical	Critical	Fixed	comcat, Xi_Zi, Kong7ych3, co2kim
AKI-10	`hashedPassword` brute force attack risk	Logical	Informational	Fixed	Xi_Zi
AKI-11	`RedEnvelopeMerkleERC721::addEnvelope` Risk of envelope being overwritten	Logical	Medium	Fixed	Kong7ych3, BradMoonUESTC, co2kim
AKI-12	Incorrect usage of `transferFrom` parameters	Logical	Medium	Fixed	Kong7ych3
AKI-13	Risk of replay envelope withdrawal	Signature Forgery or Replay	Critical	Fixed	Kong7ych3, zzzix
AKI-14	Risk of open envelope being front-run	Race condition	Critical	Acknowledged	Kong7ych3, BradMoonUESTC
AKI-15	Redundant payable label	Code Style	Informational	Fixed	Kong7ych3
AKI-16	ERC20 token transfer optimization	Gas Optimization	Informational	Fixed	Kong7ych3
AKI-17	Double payment issue	Logical	Critical	Fixed	Kong7ych3, co2kim

AKI-18	NFT compatibility issue where ERC721Enumerable is not implemented	Logical	Low	Fixed	Kong7ych3
AKI-19	Gas optimization for `totalSupply` loop	Gas Optimization	Low	Acknowledged	Kong7ych3
AKI-20	Normal functions may fail due to front-running attacks	Logical	Low	Fixed	BradMoonUESTC
AKI-21	Potential reentrancy risk, which may cause unfair envelope funds	Reentrancy	Medium	Fixed	BradMoonUESTC
AKI-22	When executing any `openEnvelope` function, the person who executes it earlier has a greater chance to obtain more benefits	Weak Sources of Randomness	Informational	Acknowledged	BradMoonUESTC

AKI-1:Gas optimization for RedEnvelopeZkERC20ChainLink

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	Informational	code/contracts/RedEnvelopeZkERC20ChainLink.sol#L19-L27	Fixed	comcat

Code

```
19:     VRFCoordinatorV2Interface COORDINATOR;
20:     // Your subscription ID.
21:     uint64 s_subscriptionId;
22:     // see https://docs.chain.link/docs/vrf-contracts/#configurations
23:     bytes32 keyHash = 0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc;
24:     uint32 callbackGasLimit = 100000;
25:     uint16 requestConfirmations = 3;
26:     uint32 numWords =  1;
27:     uint256 kThirtyDays = 30 * 86400;
```

Description

comcat : since the following global params will not be changed, it should be either constant or immutable to save gas:

```
    VRFCoordinatorV2Interface COORDINATOR;
    // Your subscription ID.
    uint64 s_subscriptionId;
    // see https://docs.chain.link/docs/vrf-contracts/#configurations
    bytes32 keyHash = 0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc;
    uint32 callbackGasLimit = 100000;
    uint16 requestConfirmations = 3;
    uint32 numWords =  1;
    uint256 kThirtyDays = 30 * 86400;
```

Recommendation

comcat : consider turn it into constant like the below:

```
    VRFCoordinatorV2Interface immutable COORDINATOR;
    uint64 constant s_subscriptionId;
    bytes32 constant keyHash = 0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc;
    uint32 constant callbackGasLimit = 100000;
    uint16 constant requestConfirmations = 3;
    uint32 constant numWords =  1;
    uint256 constant kThirtyDays = 30 * 86400;
```

Client Response

Fixed.

AKI-2:Envelopes should be marked to prevent multiple claims

Category	Severity	Code Reference	Status	Contributor
Logical	Critical	code/contracts/RedEnvelopeMerkle.sol#L14-L20 code/contracts/RedEnvelopeERC20String.sol#L21 code/contracts/CampaignEnvelopeERC20.sol#L22 code/contracts/PullPaymentERC20Envelopes.sol#L45	Fixed	comcat, Kong7ych3

Code

```
14:     function returnEnvelope(string calldata envelopeID) public {
15:         MerkleEnvelope storage env = idToEnvelopes[envelopeID];
16:         require(env.balance > 0, "Balance should be larger than zero");
17:         require(env.creator == msg.sender, "We will only return to the creator!");
18:         address payable receiver = payable(env.creator);
19:         receiver.call{value: env.balance}("");
20:     }

21:     function returnEnvelope(string memory envelopeID) public onlyOwner {

22:         require(envERC20.env.balance > 0, "balance cannot be zero");

45:     SafeERC20.safeTransferFrom(envelope.token, address(this),  msg.sender, envelope.value);
```

Description

comcat : RedEnvelopeMerkle supports everyone to create an envelope. once you created an envelope, you are qualified to call the `returnEnvelope` function. it supposed to return the ETH you send in, however, it miss to check whether you already claimed. so that you can basically call the `returnEnvelope` multiple times as you want to drain all the ETH stored inside the contract.

you may refer to the following POC:

```
    function rekt3() public {
        uint64[] memory hashedPassword = new uint64[](1);
        redEnvelope.addEnvelope{value: 1 ether}("1",uint16(1),uint256(0),hashedPassword,uint32(8));
        require(address(this).balance == 0, "start");
        vm.warp(block.timestamp + 86400);
        redEnvelope.returnEnvelope("1");
        require(address(this).balance == 1, "start");
        redEnvelope.returnEnvelope("1");
        require(address(this).balance == 2, "2nd");
        redEnvelope.returnEnvelope("1");
        require(address(this).balance == 3, "3rd");

    }
    receive() external payable{}
```

Kong7ych3 : In the RedEnvelopeMerkle contract, the envelope creator can retrieve the remaining funds in the envelope through the returnEnvelope function, but the `idToEnvelopes[envelopeID]` is not cleared after returnEnvelope is completed, which will cause the envelope creator to repeatedly call the returnEnvelope function to exhaust all funds in the contract.

Kong7ych3 : In the PullPaymentERC20Envelopes contract, the owner can retrieve the remaining funds in the envelope through the reclaimEnvelope function, but does not check whether the envelope is in a suspended state, and does not set `envelope.value` to 0. Not only will this allow the owner to use the same envelopeID to deplete the funds in the contract, it will also cause the user to still withdraw funds using the same envelopeID via the withdrawal function, even though the owner has performed a reclaimEnvelope operation on this envelopeID.

Kong7ych3 : In the CampaignEnvelopeERC20 contract, the returnEnvelope function is used to refund the envelope creator, but the creator's `envERC20.env.balance` is not set to 0 after the refund. Although this function can only be called by the owner role, there is no guarantee that a refund will be issued to the same envelopeID multiple times due to some unknown issue.

The same is true for the returnEnvelope function of the RedEnvelopeERC20String contract.

Recommendation

comcat : check whether a user has already called `returnEnvelop` function, basically delete the corresponding envelope.

```
    function returnEnvelope(string calldata envelopeID) public {
        MerkleEnvelope storage env = idToEnvelopes[envelopeID];
        require(env.balance > 0, "Balance should be larger than zero");
        require(env.creator == msg.sender, "We will only return to the creator!");
        address payable receiver = payable(env.creator);
        uint256 oldBalance = env.balance;
        delete idToEnvelopes[envelopeID];
        receiver.call{value: oldBalance}("");
    }
```

Kong7ych3 : It is recommended to clear idToEnvelopes[envelopeID] after completing the returnEnvelope operation.
Kong7ych3 : It is recommended to check whether the envelope is in a paused state when performing a reclaimEnvelope operation, and `envelope.value` must be set to 0.
Kong7ych3 : It is recommended to set `envERC20.env.balance` to 0 in the returnEnvelope function before executing the refund.

Client Response

Fixed.

AKI-3: `PullPaymentERC1155Envelopes::openEnvelope` should transfer tokens out

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	code/contracts/PullPaymentERC1155Envelope.sol#L58-L90	Mitigated	comcat, Kong7ych3, zzzix

Code

```
58:       bytes calldata signature,
59:       string calldata envelopeID,
60:       bytes32[] calldata proof,
61:       bytes32 leaf,
62:       uint256 id
63:   ) public nonReentrant {
64:     require(
65:         idToEnvelopes[envelopeID].amount.length > 0,
66:         "Envelope cannot be empty"
67:     );
68:     require(recover(signature, leaf), "signature does not seem to be provided by signer");
69:     PullPaymentERC1155Envelope storage currentEnv = idToEnvelopes[envelopeID];
70:
71:     // First check if the password has been claimed
72:     uint256 bitarrayLen = currentEnv.env.isPasswordClaimed.length;
73:     uint32 idx = uint32(uint256(leaf) % bitarrayLen);
74:     uint32 bitsetIdx = idx / 8;
75:     uint8 positionInBitset = uint8(idx % 8);
76:     uint8 curBitSet = currentEnv.env.isPasswordClaimed[bitsetIdx];
77:     require(curBitSet.bit(positionInBitset) == 0, "password already used!");
78:
79:     // Now check if it is a valid password
80:     bool isUnclaimed = MerkleProof.verify(
81:         proof,
82:         currentEnv.env.unclaimedPasswordsAndAmount,
83:         keccak256(abi.encode(msg.sender, id, leaf))
84:     );
85:     require(isUnclaimed, "password need to be valid!");
86:
87:     // claim the password
88:     currentEnv.env.isPasswordClaimed[bitsetIdx].setBit(positionInBitset);
89:   }
90:
```

Description

comcat : inside the `PullPaymentERC1155Envelopes openEnvelope` function, it checks all requirements and marked the password used, but it doesn't transfer the token out. compared with `PullPaymentERC20Envelopes.withdrawal` function, we can see that it should transfer tokens out to the receiver who open the envelope. however, it doesn't transfer anything out to receiver.
Kong7ych3 : In the PullPaymentERC1155Envelopes contract, the user can call the openEnvelope function to pass in the signature and Merkle proof for verification, but openEnvelope does not transfer ERC1155 tokens to the verified user.
zzzix : In the `PullPaymentERC1155Envelopes` contract `openEnvelope` function, it does not transfer ERC1155 tokens to the verified receiver.

Recommendation

comcat : add `token.safeBatchTransferFrom` to transfer token out to receiver.

```
    function openEnvelope(
        bytes calldata signature,
        string calldata envelopeID,
        bytes32[] calldata proof,
        bytes32 leaf,
        uint256 id
    ) public nonReentrant {
        ...
        token.safeBatchTransferFrom(address(this), msg.sender, id, amount, "");
    }
```

Kong7ych3 : If the design is not expected, it is recommended to increase the logic of issuing 1155 tokens.

Client Response

Need to add test for it still, and the logic is still incorrect.

AKI-4:Lack of support for fee-on-transfer tokens

Category	Severity	Code Reference	Status	Contributor
Logical	Low	code/contracts/PullPaymentERC20Envelopes.sol#L23-L29 code/contracts/CampaignEnvelopeERC20.sol#L29-L48 code/contracts/AkiTreasury.sol#L40-L54	Fixed	comcat

Code

```
23:  function insertEnvelope(
24:    string calldata envelopeID,
25:    address tokenAddr,
26:    uint256 value,
27:    bytes32 hashedMerkelRoot,
28:    uint32 bitarraySize
29:  ) public nonReentrant onlyOwner {

29:    function addEnvelope(uint64 envelopeID, address tokenAddr, uint256 value, uint16
numParticipants, uint8 passLength, uint256 minPerOpen, uint64[] memory hashedPassword) payable
public {
30:        require(idToEnvelopes[envelopeID].env.balance == 0, "balance not zero");
31:        require(value > 0, "Trying to create zero balance envelope");
32:        validateMinPerOpen(value, minPerOpen, numParticipants);
33:
34:        // First try to transfer the ERC20 token
35:        IERC20 token = IERC20(tokenAddr);
36:        SafeERC20.safeTransferFrom(token, msg.sender, address(this), value);
37:
38:        ERC20Envelope storage envelope = idToEnvelopes[envelopeID];
39:        envelope.env.minPerOpen = minPerOpen;
40:        envelope.env.numParticipants = numParticipants;
41:        envelope.env.creator = msg.sender;
42:        for (uint i=0; i < hashedPassword.length; i++) {
43:            envelope.env.passwords[hashedPassword[i]] = initStatus();
44:        }
45:        envelope.env.balance = value;
46:        envelope.token = token;
47:        envelope.env.passLength = passLength;
48:    }

40:  function addPayment(
41:    IERC20 token,
42:    uint256 amount,
43:    uint256 rewardStartTime,
44:    uint256 rewardEndTime
45:  ) public onlyOwner {
46:    uint256 totalShares = treasuryShares_;
47:    for (uint256 i = 0; i < rewardContracts_.length; i++) {
48:      totalShares += rewardContracts_[i].shares;
49:    }
50:
```

```
51:     uint256 totalSent = 0;
52:     SafeERC20.safeTransferFrom(token, msg.sender, address(this), amount);
53:
54:     for (uint256 i = 0; i < rewardContracts_.length; i++) {
```

Description

comcat : for the fee on transfer tokens, it will deduct the fee during transfer. so basically, the real amount the receiver received is less than the sender sent. However, for the `PullPaymentERC20Envelopes, CampaignEnvelopeERC20`, it doesn't check the real amount received from the envelop creator, just store the value that it send. it will cause a problem, which is that the last user who wish to open the envelope will always failed due to the `insufficient amount` error.

comcat : inside the `addPayment` function, it use the `ERC20.safeTransferFrom` function to transfer tokens between owner and treasury contract. however, the way it transfer tokens doesn't comply with tokens, whose transfer will deduct the fees.

Recommendation

comcat : consider use the snapshot method to check the real amount received.

```
    function addEnvelope(
        uint64 envelopeID,
        address tokenAddr,
        uint256 value,
        uint16 numParticipants,
        uint8 passLength,
        uint256 minPerOpen,
        uint64[] memory hashedPassword
    ) public payable {
        ...
        IERC20 token = IERC20(tokenAddr);
        uint balanceBefore = IERC20(token).balanceOf(address(this));
        SafeERC20.safeTransferFrom(token, msg.sender, address(this), value);
        uint amount = IERC20(token).balanceOf(address(this)) - balanceBefore;
        ...
        envelope.env.balance = amount;
        ...
    }
    function insertEnvelope(
        string calldata envelopeID,
        address tokenAddr,
        uint256 value,
        bytes32 hashedMerkelRoot,
        uint32 bitarraySize
    ) public nonReentrant onlyOwner {
        ...
        IERC20 token = IERC20(tokenAddr);
        uint balanceBefore = IERC20(token).balanceOf(address(this));
        SafeERC20.safeTransferFrom(token, msg.sender, address(this), value);
        uint amount = IERC20(token).balanceOf(address(this)) - balanceBefore;
        ...
        envelope.value = amount;
    }
```

comcat : use snapshot method to measure token received amount.

```
    function addPayment(
        IERC20 token,
        uint256 amount,
        uint256 rewardStartTime,
        uint256 rewardEndTime
    ) public onlyOwner {
        uint256 totalShares = treasuryShares_;
        for (uint256 i = 0; i < rewardContracts_.length; i++) {
            totalShares += rewardContracts_[i].shares;
        }

        uint256 totalSent = 0;
        uint256 beforeAmount = ERC20Like(token).balanceOf(address(this));
        SafeERC20.safeTransferFrom(token, msg.sender, address(this), amount);
       uint256 afterAmount = ERC20Like(token).balanceOf(address(this));
       uint amountInput = afterAmount - beforeAmount;
       ...
       // remaining will be sent to the owner address
       uint dust = ERC20Like(token).balanceOf(address(this));
       ERC20Like(token).transfer(owner(), dust);
```

Client Response

Fixed.

AKI-5:Use of weak pseudo-random number generator

Category	Severity	Code Reference	Status	Contributor
Weak Sources of Randomness	Medium	code/contracts/RedEnvelopeMerkleERC721.sol#L10-L13 code/contracts/RedEnvelopeMerkleERC721.sol#L93 code/contracts/IEnvelope.sol#L260-L273 code/contracts/IEnvelope.sol#L284	Acknowledged	comcat, Kong7ych3, BradMoonUESTC

Code

```
10:function random(uint32 number) view returns(uint32){
11:    return uint32(uint256(keccak256(abi.encodePacked(block.timestamp,block.difficulty,
12:    msg.sender)))) % number;
13:}

93:        uint32 randIdx = random(uint32(currentEnv.tokenIDs.length));

260:    function getRand(address receiver) internal virtual returns (uint16) {
261:        // we generate a psuedorandom number. The cast here is basicalluy the same as mod
262:        // https://ethereum.stackexchange.com/questions/100029/how-is-uint8-calculated-from-a-uint256-conversion-in-solidity
263:        return uint16(
264:            uint256(
265:                keccak256(
266:                    abi.encodePacked(
267:                        block.difficulty,
268:                        block.timestamp
269:                    )
270:                )
271:            )
272:        );
273:    }

284:        uint16 rand1K = rand % 1000;
```

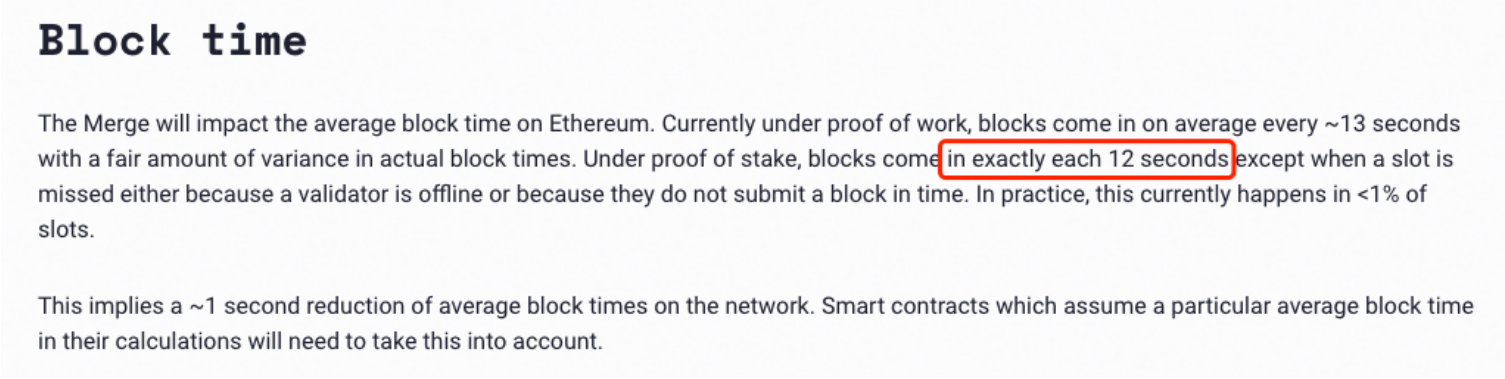
Description

comcat : the `getRand` function use the keccak of block.difficulty and block.timestamp to get the randomness, however, after merge this way can not work properly. since ETH2.0 has a fixed slot interval, namely 12 seconds. as for the block.difficulty, since ETH2.0, the block.difficulty is 0, but according to EIP-4399, it will return a random number which reflect the ETH2.0.

However, the randomness is totally calculated on chain, which is exploitable. for example, the hacker can deploy a contract, which implement the same way to get random, then the hacker use the flashbots to send a bundle, inside which requires the rand1K greater than 990. because the bundle in flashbots can only be included when all tx inside the bundle success. so the hacker can rekt the randomness without risk. you may consider the following POC:

```
function rekt2() public {
        uint16 random = uint16(uint256(keccak256(abi.encodePacked(block.difficulty,
block.timestamp))));
        uint rand1K = random % 1000;
        require(rand1K > 990, "must be greater than 990");
        ...
        envelope.openEnvelope(...)
    }
```

you may refer to the following links: https://blog.ethereum.org/2021/11/29/how-the-merge-impacts-app-layer https://eips.ethereum.org/EIPS/eip-4399



Block time

The Merge will impact the average block time on Ethereum. Currently under proof of work, blocks come in on average every ~13 seconds with a fair amount of variance in actual block times. Under proof of stake, blocks come in exactly each 12 seconds except when a slot is missed either because a validator is offline or because they do not submit a block in time. In practice, this currently happens in <1% of slots.

This implies a ~1 second reduction of average block times on the network. Smart contracts which assume a particular average block time in their calculations will need to take this into account.

Kong7ych3 : During the openEnvelope operation, if the numParticipants of this envelope is greater than 1, it will calculate how many random rewards the user can get through getMoneyThisOpen. The random number source is obtained through the getRand function, but the getRand function only uses the `block.difficulty` and `block.timestamp` of the previous block as the random number.

EIP4399 proposes that the value of `block.difficulty` will be changed to prevrandao to ensure compatibility after the Ethereum merge. In the PoS world, this value is produced by the beacon chain and is used to help determine who the next set of validators are in the block proposal and attestation process. The “prev” part of prevrandao is derived from its value being the previous block’s outputted randao value, and validators who are chosen to propose blocks will be able to know the value of prevrandao while selecting the transactions to construct their blocks out of.

So using the current block's `block.difficulty` as the random number seed is easy to predict.

The same is true for the random function in RedEnvelopeMerkleERC721.sol

BradMoonUESTC : The `rand1K` and `rand` based on block data, are very likely to be malicious use through MEV Also,

the range of `rand1K` is 0-999 and the number between 0-535 are more likely to generated because the `rand` is between 0-65535, attacker can precalculate the number of rand to make `moneyThisOpen` larger

Recommendation

comcat : - use chainlink's VRF as the source of randomness instead. you may refer to the following link: https://docs.chain.link/docs/vrf/v2/introduction/

Kong7ych3 : Best practice is to use the VRF provided by ChainLink as the source of random numbers, but we know this can be costly relative to the reward value per envelope, luckily we have other options.

We can also use `blcok.hash` or `block.difficulty` of future blocks as the source of random numbers. When the user performs the openEnvelope operation, the lottery will not be drawn immediately, but a sufficiently secure future block will be confirmed as the random number source, and the random number can only be obtained when this block is reached. This will avoid the risk of current validators doing evil to a large extent, and the cost is low.

Or you can use the random number source built on RANDAO proposed by EIP4399 https://eips.ethereum.org/EIPS/eip-4399#tips-for-application-developers

BradMoonUESTC : use chainlink oracle random number

Client Response

Acknowledged. Not fixed, I have another implementation with VRF.

AKI-6:Lack of checks for user input parameters

Category	Severity	Code Reference	Status	Contributor
Logical	Informational	code/contracts/AkiRewardSplitter.sol#L36-L38 code/contracts/AkiTreasury.sol#L40-L45 code/contracts/AkiRewardSplitter.sol#L97-L98	Fixed	comcat, BradMoonUESTC

Code

```
36:  function setTreasuryAddress(address addr) public onlyOwner {
37:    treasuryAddress_ = addr;
38:  }

40:  function addPayment(
41:    IERC20 token,
42:    uint256 amount,
43:    uint256 rewardStartTime,
44:    uint256 rewardEndTime
45:  ) public onlyOwner {

97:      env.rewardStartTime = rewardStartTime;
98:      env.rewardEndTime = rewardEndTime;
```

Description

comcat : the AkiTreasury.addPayment should add sanity check for the input params. By now, it doesn't check the input params, and accept as it is.
BradMoonUESTC : Param `addr` need to check if it's zero address
BradMoonUESTC : When `rewardStartTime` is larger than `rewardEndTime`, the Payment won't work

Recommendation

comcat : basically the amount should greater than 0, the rewardStartTime should greater than now(), the rewardEndTime should greater than rewardStartTime.

```
    function addPayment(
        IERC20 token,
        uint256 amount,
        uint256 rewardStartTime,
        uint256 rewardEndTime
    ) public onlyOwner {
        require(amount > 0, "can not add 0 tokens");
        require(rewardStartTime >= block.timestamp && rewardEndTime > rewardStartTime, "misconfigure
d reward period");
        require(isWhiteListed[token], "the token should be whitelisted");
       ...
}
```

BradMoonUESTC : Add logic of `require(addr!=address(0))`
BradMoonUESTC : Add logic of `require(rewardStartTime<rewardEndTime)`

Client Response

Fixed, including whitelist token, but the timestamp would be earlier than the block timestamp.

AKI-7:Multiple contracts are missing necessary events

Category	Severity	Code Reference	Status	Contributor
Code Style	Informational	code/contracts/AkiTreasury.sol#L20-L40 code/contracts/AkiBadge.sol#L20-L37 code/contracts/AkiRewardSplitter.sol#L36-L44	Mitigated	comcat

Code

```
20:  function setTreasuryShares(
21:    uint256 shares
22:  ) public onlyOwner {
23:    treasuryShares_ = shares;
24:  }
25:
26:  function setRewardInfo(
27:    uint256 idx,
28:    address addr,
29:    uint256 shares,
30:    address nftAddress
31:  ) public onlyOwner {
32:    while (rewardContracts_.length < idx + 1) {
33:      rewardContracts_.push();
34:    }
35:    rewardContracts_[idx].addr = addr;
36:    rewardContracts_[idx].shares = shares;
37:    rewardContracts_[idx].nftAddress = nftAddress;
38:  }
39:
40:  function addPayment(
```

```
20:    function setBaseURI(string calldata uri) public onlyOwner {
21:       baseURI_ = uri;
22:    }
23:
24:    function _baseURI() internal view override returns (string memory) {
25:        return baseURI_;
26:    }
27:
28:    // FIXME: these need to be onlyOnwer before shipped to prod
29:    function setSubscription(uint256 sub) public {
30:        subscription_ = sub;
31:    }
32:
33:    function setIsTransferrable(bool flag) public onlyOwner {
34:        isTransferrable_ = flag;
35:    }
36:
37:    function setAkiOracle(address oracle) public {
```

```
36:  function setTreasuryAddress(address addr) public onlyOwner {
```

```
37:     treasuryAddress_ = addr;
38:   }
39:
40:   function setTokenLevelInfo(
41:     address nftAddress,
42:     address[] calldata payees,
43:     uint64[] calldata shares
44:   ) public onlyOwner {
```

Description

comcat : Miss events for the AkiTreasury contract, especially for the following configure function:

```
function setTreasuryShares(uint256 shares) public
function setRewardInfo(uint256 idx, address addr, uint256 shares, address nftAddress)
function addPayment(
        IERC20 token,
        uint256 amount,
        uint256 rewardStartTime,
        uint256 rewardEndTime
    ) public onlyOwner
```

comcat : for the `AkiBadge` contract, it missed events for the following Admin's configure function:

```
function setBaseURI(string calldata uri) public
function setSubscription(uint256 sub) public
function setIsTransferrable(bool flag) public
function setAkiOracle(address oracle) public
```

comcat : for `AkiRewardSplitter` contract, it miss necessary events for the following functions:

```
function setTreasuryAddress(address addr) public
function setTokenLevelInfo(...)
```

Recommendation

comcat : emit corresponding events;

```
event TreasuryShares(uint256 indexed shares);
event RewardInfo(uint256 index idx, address index addr, uint256 shares, address indexed nftAddress);
event AddPayment(address indexed token, uint256 amount, uint256 indexed rewardStartTime, uint256
indexed rewardEndTime);
```

comcat : add corresponding events

```
event BaseURISet(string uri);
event SubscriptionSet(uint256 indexed sub);
event IsTransferrable(bool indexed flag);
event AkiOracle(address indexed oracle);
```

comcat : consider add the corresponding events for better monitor. you may refer to the following events:

```
event TreasuryAddress(address indexed addr);
function setTreasuryAddress(address addr) public onlyOwner {
        treasuryAddress_ = addr;
        emit TreasuryAddress(addr);
    }
 event TokenLevelInfo(address indexed nftAddress, address[] payees,uint64 shares);
 function setTokenLevelInfo(
        address nftAddress,
        address[] calldata payees,
        uint64[] calldata shares
    ) public onlyOwner {
        require(payees.length == shares.length, "Need to have same payment length!");
        ...
        emit TokenLevelInfo(nftAddress,payees,shares);
}
```

Client Response

Fixed some.

AKI-8:Critical parameters modification lacks permission control

Category	Severity	Code Reference	Status	Contributor
Logical	Critical	code/contracts/AkiBadge.sol#L29-L31 code/contracts/AkiBadge.sol#L29 code/contracts/AkiBadge.sol#L37-L39 code/contracts/AkiBadge.sol#L37 code/contracts/AkiBadge.sol#L53-L60	Fixed	comcat, Xi_Zi, Kong7ych3

Code

```
29:     function setSubscription(uint256 sub) public {
30:         subscription_ = sub;
31:     }

37:     function setAkiOracle(address oracle) public {
38:         akiOracle_ = oracle;
39:     }

53:     function safeMintWithVerify(
54:         uint256 tokenId,
55:         bytes32[] calldata proof
56:     ) public {
57:         AkiOracleMVP oracle = AkiOracleMVP(akiOracle_);
58:         require(oracle.verify(subscription_, proof, abi.encode(msg.sender, tokenId)), "Not verifi
ed!");
59:         _safeMint(msg.sender, tokenId);
60:     }
```

Description

comcat : the `setSubscription` function should only be called by the owner, because the subscription params get called inside the `safeMintWithVerify` function. A malicious user can call the `setSubscription` function to set the `subscription_`. by doing this, the user can manually choose which `snark` to be used for verification inside oracle.verify function. as long as the attacker can manually choose, the attacker may pass the `oracle.verify` and mint a badge
Xi_Zi : Function `setAkiOracle` modifier is public, hence anyone can set `akiOracle_`. `subscription_` in the function `safeMintWithVerify` may be affected as a factor in `oracle.verify`.
Xi_Zi : Function `setAkiOracle` modifier for public, anyone can set `akiOracle_`. By introducing an incorrect `akiOracle_` in the function `safeMintWithVerify`, the attacker is able to change the `verify` function in Oracle to a view function for require verification, and finally pass `_safeMint` to the attacker mint.
Kong7ych3 : In the AkiBadge contract, the user can mint tokens through the safeMintWithVerify function, which will check whether the Merkle proof passed in by the user is valid through the verify function of the `akiOracle_` contract, and can mint tokens only after passing the check. But any user can set the `subscription_` and `akiOracle_` parameters through the setSubscription and setAkiOracle functions. If the `akiOracle_` contract is designated as malicious and the verify function always returns true, this will make the Merkle proof check useless.

Recommendation

comcat : add access control for the `AkiBadge.setSubscription` function

```
    function setSubscription(uint256 sub) public  onlyOwner {
        subscription_ = sub;
    }
```

Xi_Zi : Change the permission of `setSubscription` to `onlyOwner`.
recommendation: Change the permission of `setSubscription` to `onlyOwner`.
Xi_Zi : Change the permission of `setAkiOracle` to `onlyOwner`.
recommendation: Change the permission of `setAkiOracle` to `onlyOwner`.
Kong7ych3 : Permission control is recommended for the setSubscription and setAkiOracle functions.

Client Response

Fixed.

AKI-9:Miss access control for the `AkiRewardSpliter::addPayment`

Category	Severity	Code Reference	Status	Contributor
Logical	Critical	code/contracts/AkiRewardSplitter.sol #L66-L100	Fixed	comcat, Xi_Zi, Kong7ych3, co2kim

Code

```
66:  function addPayment(
67:    address from,
68:    IERC20 token,
69:    uint256 amount,
70:    address nftAddress,
71:    uint256 rewardStartTime,
72:    uint256 rewardEndTime
73:  ) public {
74:    require(from == owner() || from == treasuryAddress_, "sender needs to be owner or treasury");
75:    IERC721Enumerable enumerator = IERC721Enumerable(nftAddress);
76:    uint256 totalTokens = enumerator.totalSupply();
77:    IERC721 nft = IERC721(nftAddress);
78:    // SafeERC20.safeApprove(token, from, amount);
79:    SafeERC20.safeTransferFrom(token, from, address(this), amount);
80:
81:    uint256 envId = envelopes.length;
82:    envelopes.push();
83:    PaymentEnvelope storage env = envelopes[envId];
84:    env.tokenAddress = token;
85:    env.amountRemains = amount;
86:
87:    mapping(address => uint64) storage levelInfo = nftToPayeeToShares[nftAddress];
88:    for (uint64 idx = 0; idx < totalTokens; idx++) {
89:      uint256 tokenID = IERC721Enumerable(nftAddress).tokenByIndex(idx);
90:      address receiver = nft.ownerOf(tokenID);
91:      uint64 level = levelInfo[receiver];
92:      if (level == 0) {
93:        level = 1;
94:      }
95:      env.payeeToShares[receiver] += level;
96:      env.totalShares += level;
97:      env.rewardStartTime = rewardStartTime;
98:      env.rewardEndTime = rewardEndTime;
99:    }
100:  }
```

Description

comcat : Basically the `addPayment` function should only be called by the `AkiTreasury`.`addPayment`, however, it missed the access control, which leads to mal-behavior to transfer from valuable tokens stored in Treasury or Owner to the `AkiRewardSpliter` by creating a fake envelop, which make the receiver the hacker himself. after that, the hacker can call the `pullPayment` to get the valuable token. you may refer to the following POC:

```
contract Hack is Test {
    AkiRewardSplitter public splitter;
    AkiTreasury public treasure;

    constructor(AkiRewardSplitter _splitter, AkiTreasury _treasure) {
        splitter = _splitter;
        treasure = _treasure;
    }

    function rekt() public {
        uint amount = ERC20Like(WETH).balanceOf(address(treasure));
        splitter.addPayment(
            address(treasure),
            WETH,
            amount,
            address(this),
            block.timestamp - 1,
            block.timestamp + 1
        );
        (uint share, uint currentAmount) = splitter.pullPaymentInfo();
        require(share == 1 && currentAmount == amount, "must be");
        splitter.pullPayment();
        require(ERC20Like(WETH).balanceOf(address(this)) == amount, "rekt");
    }
    function totalSupply() external view returns (uint) {
        return 1;
    }
    function tokenByIndex(uint) external view returns (uint) {
        return 0;
    }
    function ownerOf(uint256) external view returns (address) {
        return address(this);
    }
}
```

Xi_Zi : 1. The function `addPayment` permissions check by `require(from == owner() || from ==`

`treasuryAddress_, "sender needs to be owner or treasury");`, but the `from` here is an external pass, that is, an attacker can bypass the check by passing in an owner or treasuryAddress_ address as the `from` parameter.

2. At #L79, tokens that are able to transfer `from` to this Contract are transferred to this Contract by `safeTransferFrom`, that is, the owner or treasuryAddress_ address tokens can be transferred to this Contract as long as they are approved to this Contract.
3. Since `nftAddress` is also an external parameter, hackers can set the address of `receiver` as the address of the attacker in #L88-L98 by using a fake NFT in #L77, and set the corresponding parameters.
4. Finally, transfer the owner or treasuryAddress_ tokens approved into the contract through the `pullPayment` function

   Kong7ych3 : In the AkiRewardSplitter contract, the AkiRewardSplitter function is used to obtain funds from the owner role or the treasury contract to create the envelope. However, any user can call this function. If the owner role approves the MAX(uint256) allowance for this contract, then malicious users can use this function to transfer funds unauthorized from the owner role to create an envelope for additional reward distribution. Since the treasury contract only approves the required number of tokens each time the envelope is created, it is not affected by this.
   co2kim : The `AkiRewardSplitter` contract `addPayment` function is critical to add the envelope. However, there is no permission check and any EOA can call this function.

Recommendation

comcat : - add access control for the AkiRewardSplitter.addPayment function, basically add the following:

```
function addPayment(
        address from,
        IERC20 token,
        uint256 amount,
        address nftAddress,
        uint256 rewardStartTime,
        uint256 rewardEndTime
    ) public {
        require(from == owner() || from == treasuryAddress_, "sender needs to be owner or treasur
y");
        require(msg.sender == owner() || msg.sender ==  treasuryAddress_, "sender needs to be owner
or treasury");
}
```

- add sanity check for the rewardEndTime and rewardStartTime, which means that the start time should not below current block.timestamp, and the endTime should be greater than start time

```
      require(rewardStartTime >= block.timestamp && rewardEndTime > rewardStartTime, "not qualify");
```

Xi_Zi : 1. Cancel the from parameter 2. Write the verification as `require(msg.sender == owner() || msg.sender == treasuryAddress_, "sender needs to be owner or treasury");` 3. Verify nftAddress or add a nft whitelist

Kong7ych3 : Only allow access control for the AkiRewardSplitter
co2kim : Confirm if this is the desired behavior, if not, add permission check such as `onlyOwner` modifier so that only owner can call this function.

Client Response

Fixed. Removed from, just check to make sure sender is treasury.

AKI-10: `hashedPassword` brute force attack risk

Category	Severity	Code Reference	Status	Contributor
Logical	Informational	code/contracts/CampaignEnvelopeERC20.sol#L29-L81	Fixed	Xi_Zi

Code

```
29:     function addEnvelope(uint64 envelopeID, address tokenAddr, uint256 value, uint16
numParticipants, uint8 passLength, uint256 minPerOpen, uint64[] memory hashedPassword) payable
public {
30:         require(idToEnvelopes[envelopeID].env.balance == 0, "balance not zero");
31:         require(value > 0, "Trying to create zero balance envelope");
32:         validateMinPerOpen(value, minPerOpen, numParticipants);
33:
34:         // First try to transfer the ERC20 token
35:         IERC20 token = IERC20(tokenAddr);
36:         SafeERC20.safeTransferFrom(token, msg.sender, address(this), value);
37:
38:         ERC20Envelope storage envelope = idToEnvelopes[envelopeID];
39:         envelope.env.minPerOpen = minPerOpen;
40:         envelope.env.numParticipants = numParticipants;
41:         envelope.env.creator = msg.sender;
42:         for (uint i=0; i < hashedPassword.length; i++) {
43:             envelope.env.passwords[hashedPassword[i]] = initStatus();
44:         }
45:         envelope.env.balance = value;
46:         envelope.token = token;
47:         envelope.env.passLength = passLength;
48:     }
49:
50:     function openEnvelope(address payable receiver, uint64 envelopeID, string memory
unhashedPassword) public {
51:         require(idToEnvelopes[envelopeID].env.balance > 0, "Envelope is empty");
52:         uint64 passInt64 = hashPassword(unhashedPassword);
53:         ERC20Envelope storage currentEnv = idToEnvelopes[envelopeID];
54:
55:         // validate the envelope
56:         Status storage passStatus = currentEnv.env.passwords[passInt64];
57:         require(passStatus.initialized, "Invalid password!");
58:         require(!passStatus.claimed, "Password is already used");
59:         require(bytes(unhashedPassword).length == currentEnv.env.passLength, "password is incorre
ct length");
60:
61:         // claim the password
62:         currentEnv.env.passwords[passInt64].claimed = true;
63:
64:         // currently withdrawl the full balance, turn this into something either true random or
psuedorandom
65:         if (currentEnv.env.numParticipants == 1) {
```

```
66:             SafeERC20.safeApprove(currentEnv.token, address(this), currentEnv.env.balance);
67:             SafeERC20.safeTransferFrom(currentEnv.token, address(this), receiver,
currentEnv.env.balance);
68:             currentEnv.env.balance = 0;
69:             return;
70:         }
71:         uint256 moneyThisOpen = getMoneyThisOpen(
72:             receiver,
73:             currentEnv.env.balance,
74:             currentEnv.env.minPerOpen,
75:             currentEnv.env.numParticipants);
76:         currentEnv.env.numParticipants--;
77:
78:         SafeERC20.safeApprove(currentEnv.token, address(this), moneyThisOpen);
79:         SafeERC20.safeTransferFrom(currentEnv.token, address(this), receiver, moneyThisOpen);
80:         currentEnv.env.balance -= moneyThisOpen;
81:     }
```

Description

Xi_Zi : When the function `addEnvelope` is called, the related parameters will be exposed on the blockchain, and the user can know the `hashedPassword` and `passLength` of the `Envelope`, and the password of string type in line with the rules can be exploded under the chain. The value can be expressed by `uint64 passInt64 = hashPassword(unhashedPassword);` Get the correct `passInt64`, and then open the Envelope to get the reward.

Recommendation

Xi_Zi : Consider limiting the minimum length and complexity of passwords to prevent brute force.

Client Response

Fixed. Required min password length.

AKI-11: `RedEnvelopeMerkleERC721::addEnvelope` Risk of envelope being overwritten

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	code/contracts/RedEnvelopeMerkleERC721.sol#L44-L67	Fixed	Kong7ych3, BradMoonUESTC, co2kim

Code

```
44:     function addEnvelope(
45:         string calldata envelopeID,
46:         bytes32 hashedMerkelRoot,
47:         uint32 bitarraySize,
48:         address erc721ContractAddress,
49:         uint256[] calldata tokenIDs
50:     ) public nonReentrant {
51:         require(tokenIDs.length > 0, "Trying to create an empty envelope!");
52:
53:         MerkleEnvelopeERC721 storage envelope = idToEnvelopes[envelopeID];
54:         envelope.creator = msg.sender;
55:         envelope.unclaimedPasswords = hashedMerkelRoot;
56:         envelope.isPasswordClaimed = new uint8[](bitarraySize/8 + 1);
57:         envelope.tokenAddress = erc721ContractAddress;
58:         envelope.tokenIDs = tokenIDs;
59:
60:         for (uint8 tokenIdx = 0; tokenIdx < tokenIDs.length; tokenIdx++) {
61:             IERC721(erc721ContractAddress).transferFrom(
62:                 msg.sender,
63:                 address(this),
64:                 tokenIDs[tokenIdx]
65:             );
66:         }
67:     }
```

Description

Kong7ych3 : In the RedEnvelopeMerkleERC721 contract, users can create new envelopes through the addEnvelope function. However, the addEnvelope function does not check whether the envelope corresponding to the envelopeID parameter passed in by the user already exists. Therefore, any user can pass in the existing envelopeID to overwrite the

old envelope, which will cause all NFTs in the contract to be permanently locked.
BradMoonUESTC : function `addEnvelope` is not check if the envelopeID is exist like other `addEnvelope` functions by check the balance if larger than 0, the attacker can use same envelopeId to rewrite the old envelope information, disable the envelope or set old envelope password or token to malicious data, can cause all funds in envelope stolen.
co2kim : contract `RedEnvelopeMerkleERC721` function `addEnvelope` does not check if the `envelopeID` exists, and the malicious user can use same `envelopeId` to over write the existing envelope data.

Recommendation

Kong7ych3 : It is recommended to check whether the envelope corresponding to envelopeID already exists when performing the addEnvelope operation.
For example: check if `idToEnvelopes[envelopeID].tokenIDs.length` is 0.
BradMoonUESTC : Add logic of `require(idToEnvelopes[envelopeID].env.balance == 0, "balance not zero");`
co2kim : Add check logic of `require(idToEnvelopes[envelopeID].tokenIDs.length == 0, "error");`

Client Response

Fixed.

AKI-12:Incorrect usage of `transferFrom` parameters

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	code/contracts/RedEnvelopeMerkleERC721.sol#L25-L29	Fixed	Kong7ych3

Code

```
25:             IERC721(env.tokenAddress).transferFrom(
26:                 msg.sender,
27:                 address(this),
28:                 env.tokenIDs[tokenIdx]
29:             );
```

Description

Kong7ych3 : In the RedEnvelopeMerkleERC721 contract, the creator of the envelope can retrieve the remaining NFTs in the envelope through the returnEnvelope function. However, when performing NFT transfers, the from and to addresses were wrongly written, resulting in the withdrawal operation becoming a recharge operation. This function will never work.
Kong7ych3 : In the AkiRewardSplitter contract, when the reward distribution has not yet started or ended, the owner role can use the returnEnvelope function to retrieve the remaining tokens in the envelopes. It uses the safeTransferFrom interface for ERC20 token transfers, but does not approve this contract, which will cause the returnEnvelope operation to always fail.

Recommendation

Kong7ych3 : When doing NFT extraction, it should be transfer from address(this) to msg.sender.
Kong7ych3 : It is recommended to use the SafeTransfer interface when transferring ERC20 tokens of this contract.Or this contract should be approved first.

Client Response

Fixed.

AKI-13:Risk of replay envelope withdrawal

Category	Severity	Code Reference	Status	Contributor
Signature Forgery or Replay	Critical	code/contracts/RedEnvelopeMerkle.sol#L59-L66 code/contracts/PullPaymentERC1155Envelope.sol#L69-L89 code/contracts/PullPaymentERC20Envelopes.sol#L73-L92 code/contracts/RedEnvelopeMerkleERC721.sol#L75-L102 code/contracts/RedEnvelopeMerkleERC1155.sol#L78-L98 code/contracts/RedEnvelopeMerkleERC20.sol#L87-L107 code/contracts/RedEnvelopeZkERC20ChainLink.sol#L158-L178	Fixed	Kong7ych3, zzzix

Code

```
59:         require(contains == 0, "password already used!");
60:
61:         // Now check if it is a valid password
62:         bool isUnclaimed = MerkleProof.verify(proof, currentEnv.unclaimedPasswords, leaf);
63:         require(isUnclaimed, "password need to be valid!");
64:
65:         // claim the password
66:         currentEnv.isPasswordClaimed[bitsetIdx].setBit(positionInBitset);

69:     PullPaymentERC1155Envelope storage currentEnv = idToEnvelopes[envelopeID];
70:
71:     // First check if the password has been claimed
72:     uint256 bitarrayLen = currentEnv.env.isPasswordClaimed.length;
73:     uint32 idx = uint32(uint256(leaf) % bitarrayLen);
74:     uint32 bitsetIdx = idx / 8;
75:     uint8 positionInBitset = uint8(idx % 8);
76:     uint8 curBitSet = currentEnv.env.isPasswordClaimed[bitsetIdx];
77:     require(curBitSet.bit(positionInBitset) == 0, "password already used!");
78:
79:     // Now check if it is a valid password
80:     bool isUnclaimed = MerkleProof.verify(
81:         proof,
82:         currentEnv.env.unclaimedPasswordsAndAmount,
83:         keccak256(abi.encode(msg.sender, id, leaf))
84:     );
85:     require(isUnclaimed, "password need to be valid!");
86:
87:     // claim the password
88:     currentEnv.env.isPasswordClaimed[bitsetIdx].setBit(positionInBitset);
89: }

73:       require(recover(signature, leaf), "signature does not seem to be provided by signer");
74:
75:       // First check if the password has been claimed
76:       uint256 bitarrayLen = currentEnv.env.isPasswordClaimed.length;
77:       uint32 idx = uint32(uint256(leaf) % bitarrayLen);
78:       uint32 bitsetIdx = idx / 8;
79:       uint8 positionInBitset = uint8(idx % 8);
80:       uint8 curBitSet = currentEnv.env.isPasswordClaimed[bitsetIdx];
81:       require(curBitSet.bit(positionInBitset) == 0, "password already used!");
```

```
82:
83:     // Now check if it is a valid password
84:     bool validAmountAndSender = MerkleProof.verify(
85:         proof,
86:         currentEnv.env.unclaimedPasswordsAndAmount,
87:         keccak256(abi.encode(msg.sender, leaf, amount))
88:     );
89:     require(validAmountAndSender, "password need to be valid!");
90:
91:     // claim the password
92:     currentEnv.env.isPasswordClaimed[bitsetIdx].setBit(positionInBitset);

75:       require(recover(signature, leaf), "signature does not seem to be provided by signer");
76:       require(idToEnvelopes[envelopeID].tokenIDs.length > 0, "Envelope cannot be empty");
77:       MerkleEnvelopeERC721 storage currentEnv = idToEnvelopes[envelopeID];
78:
79:       // First check if the password has been claimed
80:       // check index of the bitset, then check the position in the bitset
81:       uint256 bitarrayLen = currentEnv.isPasswordClaimed.length;
82:       uint32 idx = uint32(uint256(leaf) % bitarrayLen);
83:       uint32 bitsetIdx = idx / 8;
84:       uint8 positionInBitset = uint8(idx % 8);
85:       uint8 curBitSet = currentEnv.isPasswordClaimed[bitsetIdx];
86:       require(curBitSet.bit(positionInBitset) == 0, "password already used!");
87:
88:       // Now check if it is a valid password
89:       bool isUnclaimed = MerkleProof.verify(proof, currentEnv.unclaimedPasswords, leaf);
90:       require(isUnclaimed, "password need to be valid!");
91:
92:       // FXIME pick a random id in the array instead of just the first
93:       uint32 randIdx = random(uint32(currentEnv.tokenIDs.length));
94:       IERC721(currentEnv.tokenAddress).transferFrom(
95:           address(this),
96:           msg.sender,
97:           currentEnv.tokenIDs[randIdx]
98:       );
99:       _burn(currentEnv.tokenIDs, randIdx);
100:
101:       // claim the password
102:       currentEnv.isPasswordClaimed[bitsetIdx].setBit(positionInBitset);
```

```
78:         require(recover(signature, leaf), "signature does not seem to be provided by signer");
79:         MerkleERC1155Envelope storage currentEnv = idToEnvelopes[envelopeID];
80:
81:         // First check if the password has been claimed
82:         uint256 bitarrayLen = currentEnv.isPasswordClaimed.length;
83:         uint32 idx = uint32(uint256(leaf) % bitarrayLen);
84:         uint32 bitsetIdx = idx / 8;
85:         uint8 positionInBitset = uint8(idx % 8);
86:         uint8 curBitSet = currentEnv.isPasswordClaimed[bitsetIdx];
87:         require(curBitSet.bit(positionInBitset) == 0, "password already used!");
88:
89:         // Now check if it is a valid password
90:         bool isUnclaimed = MerkleProof.verify(
91:             proof,
92:             currentEnv.unclaimedPasswords,
93:             leaf
94:         );
95:         require(isUnclaimed, "password need to be valid!");
96:
97:         // claim the password
98:         currentEnv.isPasswordClaimed[bitsetIdx].setBit(positionInBitset);

87:         require(recover(signature, leaf), "signature does not seem to be provided by signer");
88:         MerkleERC20Envelope storage currentEnv = idToEnvelopes[envelopeID];
89:
90:         // First check if the password has been claimed
91:         uint256 bitarrayLen = currentEnv.isPasswordClaimed.length;
92:         uint32 idx = uint32(uint256(leaf) % bitarrayLen);
93:         uint32 bitsetIdx = idx / 8;
94:         uint8 positionInBitset = uint8(idx % 8);
95:         uint8 curBitSet = currentEnv.isPasswordClaimed[bitsetIdx];
96:         require(curBitSet.bit(positionInBitset) == 0, "password already used!");
97:
98:         // Now check if it is a valid password
99:         bool isUnclaimed = MerkleProof.verify(
100:             proof,
101:             currentEnv.unclaimedPasswords,
102:             leaf
103:          );
104:          require(isUnclaimed, "password need to be valid!");
105:
```

```
106:         // claim the password
107:         currentEnv.isPasswordClaimed[bitsetIdx].setBit(positionInBitset);

158:         require(recover(signature, leaf), "signature does not seem to be provided by signer");
159:         MerkleERC20Envelope storage currentEnv = idToEnvelopes[envelopeID];
160:
161:         // First check if the password has been claimed
162:         uint256 bitarrayLen = currentEnv.isPasswordClaimed.length;
163:         uint32 idx = uint32(uint256(leaf) % bitarrayLen);
164:         uint32 bitsetIdx = idx / 8;
165:         uint8 positionInBitset = uint8(idx % 8);
166:         uint8 curBitSet = currentEnv.isPasswordClaimed[bitsetIdx];
167:         require(curBitSet.bit(positionInBitset) == 0, "password already used!");
168:
169:         // Now check if it is a valid password
170:         bool isUnclaimed = MerkleProof.verify(
171:             proof,
172:             currentEnv.unclaimedPasswords,
173:             leaf
174:         );
175:         require(isUnclaimed, "password need to be valid!");
176:
177:         // claim the password
178:         currentEnv.isPasswordClaimed[bitsetIdx].setBit(positionInBitset);
```

Description

Kong7ych3 : In the PullPaymentERC20Envelopes contract, the user can use the signature and Merkle proof through the withdrawal function to withdraw the funds in the envelope. It will first check whether `curBitSet.bit(positionInBitset)` is 0, and set isPasswordClaimed through `currentEnv.env.isPasswordClaimed[bitsetIdx].setBit(positionInBitset)` after completing Merkle proof verification to avoid replay issue. But the setBit function is a pure function `currentEnv.env.isPasswordClaimed[bitsetIdx].setBit(positionInBitset)` will only return a value of type uint256 , so isPasswordClaimed will always be 0. This will lead to replay issues where the user can deplete the funds in the contract by repeatedly calling the withdrawal function to submit the same parameters.

The same is true for the openEnvelope function in RedEnvelopeMerkle, RedEnvelopeMerkleERC20, RedEnvelopeMerkleERC721, RedEnvelopeMerkleERC1155 and RedEnvelopeZkERC20ChainLink contract.

zzzix : The `setBit` function is a pure function and it does not alter any contract state. In the `PullPaymentERC20Envelopes` contract, there are multiple locations where `currentEnv.isPasswordClaimed[bitsetIdx].setBit(positionInBitset);` is used to try to set the bit to claim the password. As the `isPasswordClaimed` state is not changed after the function call, user can call the function again.

Recommendation

Kong7ych3 : It should be to assign the return value of setBit to isPasswordClaimed. However, since the return value of setBit is uint256, and the type of isPasswordClaimed is uint8, the remainder operation is still required to obtain the final value required by isPasswordClaimed

zzzix : The `isPasswordClaimed` state should be updated in the `setBit` function.

Client Response

Fixed.

AKI-14:Risk of open envelope being front-run

Category	Severity	Code Reference	Status	Contributor
Race condition	Critical	code/contracts/RedEnvelope.sol#L45 code/contracts/CampaignEnvelopeERC20.sol#L50 code/contracts/RedEnvelopeERC20String.sol#L56 code/contracts/RedEnvelopeERC20.sol#L61	Acknowledged	Kong7ych3, BradMoonUESTC

Code

```
45:    function openEnvelope(bytes calldata signature, uint64 envelopeID, string calldata
unhashedPassword) public nonReentrant {

50:    function openEnvelope(address payable receiver, uint64 envelopeID, string memory
unhashedPassword) public {

56:    function openEnvelope(address payable receiver, string memory envelopeID, string memory
unhashedPassword) public {

61:    function openEnvelope(bytes calldata signature, uint64 envelopeID, string calldata
unhashedPassword) public nonReentrant {
```

Description

Kong7ych3 : In the CampaignEnvelopeERC20 contract, users can pass in the correct unhashedPassword through the openEnvelope function to claim rewards. But unfortunately, the current MEV front-run attack is serious. Once the MEV bot confirms that the openEnvelope transaction is profitable, it will use this unhashedPassword to claim the reward first. And ordinary users will never be able to compete with such MEV bots.

Note that this happens almost 100% on the ETH and BSC chains!

The same is true for the openEnvelope function in the RedEnvelope contract.

The same is true for the openEnvelope function in the RedEnvelopeERC20 contract.

The same is true for the openEnvelope function in the RedEnvelopeERC20String contract.

BradMoonUESTC : Due to there is lack of signature check in `openEnvelope`, the attacker can monitor all the transaction to `RedEnvelopeERC20String.openEnvelope()` and copy the transaction and modify the input param `receiver` to attacker's account and execute the transaction by raise the gas fee.

Enventually, attacker can get all envelope funds to attacker's account

Recommendation

Kong7ych3 : It is recommended using a Merkle certificate or owner's signature containing the msg.sender address to claim rewards. This will eliminate the risk of MEV preemptively claiming rewards.

BradMoonUESTC : Add Signature Check

Client Response

Already done it with the recover function.

AKI-15:Redundant payable label

Category	Severity	Code Reference	Status	Contributor
Code Style	Informational	code/contracts/CampaignEnvelopeERC20.sol#L23 code/contracts/RedEnvelopeERC20String.sol#L24 code/contracts/RedEnvelopeERC20.sol#L26 code/contracts/CampaignEnvelopeERC20.sol#L29 code/contracts/RedEnvelopeMerkleERC20.sol#L38 code/contracts/CampaignEnvelopeERC20.sol#L50 code/contracts/RedEnvelopeZkERC20ChainLink.sol#L100	Fixed	Kong7ych3

Code

```
23:         address payable receiver = payable(envERC20.env.creator);

24:         address payable receiver = payable(envERC20.env.creator);

26:         address receiver = payable(envERC20.env.creator);

29:     function addEnvelope(uint64 envelopeID, address tokenAddr, uint256 value, uint16
numParticipants, uint8 passLength, uint256 minPerOpen, uint64[] memory hashedPassword) payable
public {

38:         address receiver = payable(env.creator);

50:     function openEnvelope(address payable receiver, uint64 envelopeID, string memory
unhashedPassword) public {

100:             address receiver = payable(env.creator);
```

Description

Kong7ych3 : In the CampaignEnvelopeERC20 contract, the owner can return funds to the creator of the envelope through the returnEnvelope function. It will first mark `envERC20.env.creator` as payable, and then transfer ERC20 tokens, but does not perform native token transfers, so the payable mark is redundant.

And in the addEnvelope function, the ERC20 token of msg.sender is transferred to this contract instead of the native token, but addEnvelope still uses the payable tag, which is redundant.

And in the openEnvelope function, the native token is not transferred to the receiver address, but only the ERC20 token is transferred to it. Therefore it is redundant to tag the receiver address with the payable tag.

The same is true for the returnEnvelope function in the RedEnvelopeERC20, RedEnvelopeERC20String, RedEnvelopeMerkleERC20 contract.

The same is true for the performUpkeep function in the RedEnvelopeZkERC20ChainLink contract.

Recommendation

Kong7ych3 : If it is not intended, it is recommended to remove redundant payable tags.

Client Response

Fixed.

AKI-16:ERC20 token transfer optimization

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	Informational	code/contracts/CampaignEnvelopeERC20.sol#L25 code/contracts/CampaignEnvelopeERC20.sol#L26 code/contracts/RedEnvelopeERC20String.sol#L26 code/contracts/RedEnvelopeERC20String.sol#L27 code/contracts/RedEnvelopeERC20.sol#L30 code/contracts/RedEnvelopeERC20.sol#L31 code/contracts/RedEnvelopeMerkleERC20.sol#L41 code/contracts/RedEnvelopeMerkleERC20.sol#L42 code/contracts/AkiTreasury.sol#L66 code/contracts/AkiTreasury.sol#L67 code/contracts/RedEnvelopeERC20String.sol#L72 code/contracts/RedEnvelopeERC20String.sol#L73 code/contracts/CampaignEnvelopeERC20.sol#L78	Fixed	Kong7ych3

		code/contracts/AkiRewardSplitter.sol#L79 code/contracts/CampaignEnvelopeERC20.sol#L79 code/contracts/RedEnvelopeERC20.sol#L81 code/contracts/RedEnvelopeERC20.sol#L82 code/contracts/RedEnvelopeERC20String.sol#L84 code/contracts/RedEnvelopeERC20String.sol#L85 code/contracts/RedEnvelopeERC20.sol#L93 code/contracts/RedEnvelopeERC20.sol#L94 code/contracts/RedEnvelopeZkERC20ChainLink.sol#L109 code/contracts/RedEnvelopeZkERC20ChainLink.sol#L110 code/contracts/RedEnvelopeMerkleERC20.sol#L112 code/contracts/RedEnvelopeMerkleERC20.sol#L113 code/contracts/RedEnvelopeMerkleERC20.sol#L127		
		code/contracts/RedEnvelopeMerkleERC20.sol#L128 code/contracts/AkiRewardSplitter.sol#L138 code/contracts/AkiRewardSplitter.sol#L142 code/contracts/RedEnvelopeZkERC20ChainLink.sol#L183 code/contracts/RedEnvelopeZkERC20ChainLink.sol#L184 code/contracts/RedEnvelopeZkERC20ChainLink.sol#L198 code/contracts/RedEnvelopeZkERC20ChainLink.sol#L199		

Code

```
25:         SafeERC20.safeApprove(token, address(this), envERC20.env.balance);

26:         SafeERC20.safeTransferFrom(token, address(this), receiver, envERC20.env.balance);

26:         SafeERC20.safeApprove(token, address(this), envERC20.env.balance);

27:         SafeERC20.safeTransferFrom(token, address(this), receiver, envERC20.env.balance);

30:         SafeERC20.safeApprove(token, address(this), oldBalance);

31:         SafeERC20.safeTransferFrom(token, address(this), receiver, oldBalance);

41:         SafeERC20.safeApprove(token, address(this), oldBalance);

42:         SafeERC20.safeTransferFrom(token, address(this), receiver, oldBalance);

66:     SafeERC20.safeApprove(token, address(this), amount - totalSent);

67:     SafeERC20.safeTransferFrom(token, address(this), owner(), amount - totalSent);

72:             SafeERC20.safeApprove(currentEnv.token, address(this), currentEnv.env.balance);

73:             SafeERC20.safeTransferFrom(currentEnv.token, address(this), receiver,
currentEnv.env.balance);

78:         SafeERC20.safeApprove(currentEnv.token, address(this), moneyThisOpen);

79:     SafeERC20.safeTransferFrom(token, from, address(this), amount);

79:         SafeERC20.safeTransferFrom(currentEnv.token, address(this), receiver, moneyThisOpen);

81:             SafeERC20.safeApprove(currentEnv.token, address(this), fullBalance);

82:             SafeERC20.safeTransferFrom(currentEnv.token, address(this), receiver, fullBalance);

84:         SafeERC20.safeApprove(currentEnv.token, address(this), moneyThisOpen);
```

```
85:         SafeERC20.safeTransferFrom(currentEnv.token, address(this), receiver, moneyThisOpen);

93:         SafeERC20.safeApprove(currentEnv.token, address(this), moneyThisOpen);

94:         SafeERC20.safeTransferFrom(currentEnv.token, address(this), receiver, moneyThisOpen);

109:             SafeERC20.safeApprove(token, address(this), oldBalance);

110:             SafeERC20.safeTransferFrom(token, address(this), receiver, oldBalance);

112:             SafeERC20.safeApprove(currentEnv.token, address(this), oldBalance);

113:             SafeERC20.safeTransferFrom(currentEnv.token, address(this), msg.sender, oldBalance);

127:         SafeERC20.safeApprove(currentEnv.token, address(this), moneyThisOpen);

128:         SafeERC20.safeTransferFrom(currentEnv.token, address(this), msg.sender, moneyThisOpen);

138:         SafeERC20.safeApprove(env.tokenAddress, address(this), currentAmount);

142:         SafeERC20.safeTransferFrom(env.tokenAddress, address(this), receiver,
info.currentAmount);

183:             SafeERC20.safeApprove(currentEnv.token, address(this), oldBalance);

184:             SafeERC20.safeTransferFrom(currentEnv.token, address(this), msg.sender, oldBalance);

198:         SafeERC20.safeApprove(currentEnv.token, address(this), moneyThisOpen);

199:         SafeERC20.safeTransferFrom(currentEnv.token, address(this), msg.sender, moneyThisOpen);
```

Description

Kong7ych3 : In the AkiRewardSplitter contract, when the returnEnvelope and pullPayment operations are performed, the contract will be approved first, and then the ERC20 tokens will be transferred from the contract to the designated user

through the safeTransferFrom interface of the SafeERC20 library. But the SafeERC20 library also has a SafeTrasnfer interface that allows direct token transfers without first approving this contract. This will save a lot of gas.

The same is true for the addPayment function in the AkiTreasury contract.

The same is true for the returnEnvelope and openEnvelope function in the CampaignEnvelopeERC20, RedEnvelopeERC20, RedEnvelopeERC20String, RedEnvelopeMerkleERC20 contract.

The same is true for the performUpkeep and openEnvelope functions in the RedEnvelopeZkERC20ChainLink contract.

Recommendation

Kong7ych3 : It is recommended to use the SafeTransfer interface when transferring ERC20 tokens of this contract.

Client Response

Fixed.

AKI-17:Double payment issue

Category	Severity	Code Reference	Status	Contributor
Logical	Critical	code/contracts/AkiRewardSplitter.sol #L129-L146	Fixed	Kong7ych3, co2kim

Code

```
129:  function pullPayment() public {
130:    address receiver = msg.sender;
131:    PaymentInfo[] memory infos = pullPaymentInfo();
132:    for (uint64 i = 0; i < envelopes.length; i++) {
133:      if (infos[i].currentAmount != 0) {
134:        PaymentEnvelope storage env = envelopes[i];
135:        PaymentInfo memory info = infos[i];
136:        uint256 currentAmount = info.share * env.amountRemains / env.totalShares;
137:        // console.log("shares", share, env.totalShares, env.amountRemains);
138:        SafeERC20.safeApprove(env.tokenAddress, address(this), currentAmount);
139:
140:        env.totalShares -= info.share;
141:        env.amountRemains -= info.currentAmount;
142:        SafeERC20.safeTransferFrom(env.tokenAddress, address(this), receiver,
info.currentAmount);
143:
144:      }
145:    }
146:  }
```

Description

Kong7ych3 : In the AkiRewardSplitter contract, users can receive token rewards through the pullPayment function, and both `env.totalShares` and `env.amountRemains` will be reduced accordingly. But the user's `env.payeeToShares` is not set to 0, which will cause the user to repeatedly call the pullPayment function to claim additional rewards until the `env.amountRemains` is exhausted.
co2kim : The `AkiRewardSplitter` contract `pullPayment` function, user's `env.payeeToShares` is not set to 0 after the `env.totalShares` and `env.amountRemains` is reducted. This means user can call the `pullPayment` function again to pull more rewards

Recommendation

Kong7ych3 : It is recommended to set the user's share to 0 after the pullPayment operation.
co2kim : Set the user's `env.payeeToShares` to 0 before the `safeTransferFrom` call.

Client Response

Fixed.

AKI-18:NFT compatibility issue where ERC721Enumerable is not implemented

Category	Severity	Code Reference	Status	Contributor
Logical	Low	code/contracts/AkiRewardSplitter.sol #L89	Fixed	Kong7ych3

Code

```
89:     uint256 tokenID = IERC721Enumerable(nftAddress).tokenByIndex(idx);
```

Description

Kong7ych3 : In the AkiRewardSplitter contract, the tokenId is obtained through the tokenByIndex function during the addPayment operation to set the envelope of the NFT holder. However, if the incoming nftAddress does not implement ERC721Enumerable, it will make it impossible to set the envelope for the holder of such NFT.

Recommendation

Kong7ych3 : It is recommended to check whether the added NFT implements ERC721Enumerable through the supportsInterface interface when performing the setRewardInfo operation. Or implement a compatible function that sets the envelope for the specified tokenId alone.

Client Response

Fixed.

AKI-19:Gas optimization for `totalSupply` loop

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	Low	code/contracts/AkiRewardSplitter.sol #L88-L99	Acknowledged	Kong7ych3

Code

```
88:     for (uint64 idx = 0; idx < totalTokens; idx++) {
89:       uint256 tokenID = IERC721Enumerable(nftAddress).tokenByIndex(idx);
90:       address receiver = nft.ownerOf(tokenID);
91:       uint64 level = levelInfo[receiver];
92:       if (level == 0) {
93:         level = 1;
94:       }
95:       env.payeeToShares[receiver] += level;
96:       env.totalShares += level;
97:       env.rewardStartTime = rewardStartTime;
98:       env.rewardEndTime = rewardEndTime;
99:     }
```

Description

Kong7ych3 : In the AkiRewardSplitter contract, when the addPayment operation is performed, all NFT token holders are obtained through a for loop to update the envelope of each user. If the total supply of NFT tokens is very large, there may be an out of gas issue due to the gas cap per block.

btw, there are still many places in the contract where there are nested for loops, which can lead to DoS

Recommendation

Kong7ych3 : If the total supply of nft tokens is too large, it is recommended to set it in batches. Add the index parameters of tokenId start and end in the addPayment function, and only modify the tokenId within the specified range each time.

```
function addPayment(
    address from,
    IERC20 token,
    uint256 amount,
    address nftAddress,
    uint256 rewardStartTime,
    uint256 rewardEndTime,
    uint256 startIndex,
    uint256 endIndex
  ) {
  ...
  for (uint64 idx = startIndex; idx < endIndex; idx++) {
    ...
    }
  }
```

Client Response

Gas optimization make sense, will add it in the future iteration. Will require quite some time.

AKI-20:Normal functions may fail due to front-running attacks

Category	Severity	Code Reference	Status	Contributor
Logical	Low	code/contracts/RedEnvelopeMerkleERC20.sol#L58	Fixed	BradMoonUESTC

Code

```
58:         require(idToEnvelopes[envelopeID].balance == 0, "balance not zero");
```

Description

BradMoonUESTC : When an attacker finds that a normal user wants to `addEnvelope`, the attacker can send a front-running attack with same `envelopeID` with a higher gas fees and small amount of tokens, attacker can add a malicious Envelope by setting the same `envelopeID`, make normal user can not pass the logic of `require(idToEnvelopes[envelopeID].env.balance == 0, "balance not zero");`
When an attacker intends to maliciously destroy the project, the Envelope function of the entire project can be disabled

Recommendation

BradMoonUESTC : Set evnvelopeID to auto increment, or raise the fund threshold of `addEnvelope`

Client Response

Added signature for addEnvelope to prevent front-run.

AKI-21:Potential reentrancy risk, which may cause unfair envelope funds

Category	Severity	Code Reference	Status	Contributor
Reentrancy	Medium	code/contracts/RedEnvelopeMerkle.sol#L82	Fixed	BradMoonUESTC

Code

```
82:         receiver.call{value: moneyThisOpen}("");
```

Description

BradMoonUESTC : The attacker may obtain improper benefits by reentrance the `openEnvlope` function

Every calculation of `moneyThisOpen` depends on `env. balance`. If the `env. balance` is larger, moneyThisOpen may be larger

If an attacker holds multiple passwords, he or she can reentrant and read dirty data because the `currentEnv.env balance -= moneyThisOpen;` is not exeuted

The error `env Balance` Data is used to obtain `moneyThisOpen` funds, which is larger than the amount of funds obtained by a single execution of `openEnvelope`

Recommendation

BradMoonUESTC : move the logic of `currentEnv.env.balance -= moneyThisOpen;` just after the logic of `currentEnv.env.numParticipants--;` or add reentrancy lock

Client Response

Fixed.

AKI-22:When executing any `openEnvelope` function, the person who executes it earlier has a greater chance to obtain more benefits

Category	Severity	Code Reference	Status	Contributor
Weak Sources of Randomness	Informational	code/contracts/IEnvelope.sol#L280	Acknowledged	BradMoonUESTC

Code

```
280:     ) public returns (uint256) {
```

Description

BradMoonUESTC : The earlier the person trades, the higher the `numParticipants` when `getMoneyThisOpen` is executed. If `rand1K` is the same, the lower the `randBalance` and the lower the maxthisopen

Recommendation

BradMoonUESTC : Reconsider the envelope distribution mechanism

Client Response

I don't have a good answer here, but the people who execute earlier get some advantage and we should be mostly ok with that. I will think about the mechanism for a bit.

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.