33AUDITS & CO.

Gaia Rewards Audit Report

Introduction

A time-boxed security review of the protocol was done by 33Audit & Company, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits as the Lead Senior Security Researcher and Bumble as the Security Researcher.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About 33 Audits & Company

33Audits LLC is an independent smart contract security researcher company and development group. We conduct audits as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @solidityauditor.

About Gaia Protocol

This report presents the findings of a smart contract audit of the Gaia Net", mostly focused on the Gaia Domain Name (GDN) System. The GDN Protocol is a decentralized domain name service built on Ethereum network that allows users to register, manage, and trade domain names as NFTs. The protocol includes features for domain registration, staking, governance, and reward distribution.

Key components of the system include:

- Domain name registration and management (GdnRegistrar, GdnRegistry)
- Payment processing and USDT integration
- Staking mechanism for domains
- Governance system with timelock controls
- Reward distribution for stakers and node operators
- Token economics with the GAIA token

Severity Definitions

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

Scope

The audit covered the following contracts at commit hash `8d25043bb1c9adc97417bec8eba16d155ab7d47a`:

Contract	Description	Link
GdnRegistrationService.sol	Main service for domain registration	View
GdnRegistrar.sol	NFT contract for domain ownership	View
GdnRegistry.sol	Core registry maintaining domain records	View
GaiaReward.sol	Reward distribution system	View
GaiaDomainStaking.sol	Domain staking mechanism	View
GaiaGovernor.sol	Governance contract	View
GaiaTimelock.sol	Timelock controller for governance	View
GDNPaymentReceiver.sol	Payment processing contract	View
GaiaTokenBase.sol	Base network implementation of the GAIA token	View

Test Coverage Analysis

Importance of Complete Test Coverage and Fuzz Testing

A comprehensive test suite is crucial for smart contract security. High test coverage helps ensure that:

1. All code paths are exercised and verified
2. Edge cases are identified and handled properly
3. Contract interactions work as expected
4. Regressions are caught early

Fuzz testing is particularly valuable for smart contracts as it:

- Automatically generates random inputs to find edge cases
- Can identify unexpected behavior that manual testing might miss
- Helps verify input validation and boundary conditions
- Can discover complex multi-step vulnerabilities

Initial Coverage Report

Below is the initial coverage report for the core contracts:

Contract	% Lines	% Statements	% Branches	% Functions
GaiaTokenBase.sol	40.00%	40.00%	25.00%	40.00%
GaiaTokenEth.sol	66.67%	50.00%	100.00%	66.67%
GaiaReward.sol	93.83%	96.51%	56.10%	83.33%
GdnRegistry.sol	100.00%	100.00%	100.00%	100.00%
GdnRegistrar.sol	100.00%	100.00%	75.00%	100.00%
GdnRegistrationService.sol	88.57%	83.56%	55.56%	100.00%

Updated Coverage Report

After implementing additional tests and fuzz testing, coverage has improved significantly:

Contract	% Lines	% Statements	% Branches	% Functions
GaiaTokenBase.sol	90.00%	83.33%	75.00%	80.00%
GaiaTokenEth.sol	100.00%	100.00%	100.00%	100.00%
GaiaReward.sol	92.94%	96.59%	56.10%	78.57%
GdnRegistry.sol	100.00%	100.00%	100.00%	100.00%
GdnRegistrar.sol	100.00%	100.00%	75.00%	100.00%
GdnRegistrationService.sol	88.57%	83.56%	55.56%	100.00%

Key Improvements

- GaiaTokenBase and GaiaTokenEth show significant improvements in all areas
- Overall branch coverage has improved but still needs attention in some contracts
- Most contracts now have >80% coverage across all metrics

Recommendations

1. Implement comprehensive test suites for GaiaDomainStaking and GaiaGovernor
2. Add fuzz testing for key functions, especially those handling user inputs
3. Increase branch coverage through additional edge case tests
4. Focus on improving function coverage in partially tested contracts

Findings Summary

ID	Title	Severity	Status

ID	Title	Severity	Status
[H-01]	Unstaking Mechanism Incorrectly Resets for Partial Withdrawals	High	Fixed
[H-02]	Potential Hash Collisions in Signature Verification	High	Fixed
[M-01]	Unbounded Loop in Domain Management Can Lead to DOS	Medium	Fixed
[M-02]	Unsafe ERC721 Minting in GDN Domain System	Medium	Fixed
[M-03]	`pausable` Function Needs to be Implemented	Medium	Fixed
[M-04]	Set Beneficiary Should be a Two Step Process	Medium	Fixed
[M-05]	Unsafe Token Transfer Pattern in Reward Claims	Medium	Fixed
[L-01]	Missing Zero Amount Validation in Reward Distribution Functions	Low	Fixed
[L-02]	Indexed String Events Emit Hashes Instead of Values	Low	Fixed
[L-03]	Unchecked Array Length Matching in Distribution Functions	Low	Fixed
[I-01]	Timestamp Checks Order: The timestamp checks are performed after creating the message hash.	Informational	Fixed

HIGH

[H-01] - Unstaking Mechanism Incorrectly Resets for Partial Withdrawals

Description

The unstaking mechanism in `GaiaDomainStaking.sol` has a critical flaw where partial withdrawals reset the unstaking state for the entire staked amount. When a user withdraws less than their requested unstake amount, the remaining tokens that should be available for withdrawal become locked again, requiring another unstake request and delay period.

```
function withdraw(string calldata gdn, uint256 amount) external {
    Domain storage domain = domains[gdn];
    StakeInfo storage stakeInfo = domain.stakers[msg.sender];

    require(stakeInfo.isUnstaking, "No unstake requested");
```

```
        require(stakeInfo.amount >= amount, "Insufficient staked amount");
        require(
            block.timestamp >= stakeInfo.unstakeRequestTime + unstakeDelay,
            "Unstake delay not met"
        );

        stakeInfo.amount -= amount;
        domain.totalStaked -= amount;
        stakeInfo.isUnstaking = false;  // @audit: Resets unstaking state for
ALL remaining tokens
        stakeInfo.unstakeRequestTime = 0;  // @audit: Resets timestamp for ALL
remaining tokens

        // ... rest of function
    }
```

Impact

- High severity as it unnecessarily locks user funds
- Forces users to wait multiple delay periods to fully withdraw their funds
- Increases gas costs due to multiple unstake/withdraw transactions

Proof of Concept

1. Alice stakes 1000 GAIA tokens in a domain
2. Alice requests to unstake 500 GAIA (`unstake(500)`)
   - Sets `isUnstaking = true`
   - Sets `unstakeRequestTime = currentTime`
3. Alice waits for the unstakeDelay period (e.g., 7 days)
4. Alice withdraws 100 GAIA (`withdraw(100)`)
   - Contract transfers 100 GAIA to Alice
   - Sets `isUnstaking = false` for ALL remaining funds
   - Sets `unstakeRequestTime = 0` for ALL remaining funds
5. Alice still has 400 GAIA from her original 500 GAIA unstake request
6. Alice must call `unstake()` again and wait another 7 days to access the remaining 400 GAIA

Fix

Track unstaking amounts separately from the unstaking state:

```
struct StakeInfo {
    string gdn;
    uint256 amount;
    uint256 lastStakeTime;
    uint256 unstakeRequestTime;
    uint256 unstakeAmount;  // Add tracking for requested unstake amount
    bool isUnstaking;
}
```

```
function withdraw(string calldata gdn, uint256 amount) external {
    Domain storage domain = domains[gdn];
    StakeInfo storage stakeInfo = domain.stakers[msg.sender];

    require(stakeInfo.isUnstaking, "No unstake requested");
    require(amount <= stakeInfo.unstakeAmount, "Amount exceeds unstake
request");
    require(stakeInfo.amount >= amount, "Insufficient staked amount");
    require(
        block.timestamp >= stakeInfo.unstakeRequestTime + unstakeDelay,
        "Unstake delay not met"
    );

    stakeInfo.amount -= amount;
    domain.totalStaked -= amount;
    stakeInfo.unstakeAmount -= amount;

    // Only reset unstaking state if all requested amount is withdrawn
    if (stakeInfo.unstakeAmount == 0) {
        stakeInfo.isUnstaking = false;
        stakeInfo.unstakeRequestTime = 0;
    }

    if (stakeInfo.amount == 0) {
        _removeStakerDomain(msg.sender, gdn);
    }

    gaiaToken.transfer(msg.sender, amount);
    emit Withdrawn(gdn, amount, msg.sender);
}

function unstake(string calldata gdn, uint256 amount) external {
    Domain storage domain = domains[gdn];
    StakeInfo storage stakeInfo = domain.stakers[msg.sender];

    require(stakeInfo.amount >= amount, "Insufficient staked amount");
    require(!stakeInfo.isUnstaking, "Already unstaking");
    require(
        block.timestamp >= stakeInfo.lastStakeTime + stakeLockPeriod,
        "Stake still locked"
    );

    stakeInfo.isUnstaking = true;
    stakeInfo.unstakeAmount = amount;  // Track requested unstake amount
    stakeInfo.unstakeRequestTime = block.timestamp;

    emit Unstaked(gdn, amount, block.timestamp, msg.sender);
}
```

Recommendation

Implement the proposed fix to track unstaking amounts separately and only reset the unstaking state when the full requested amount has been withdrawn. This allows users to withdraw their unstaked tokens in multiple transactions without requiring additional delay periods.

[H-02] - Potential Hash Collisions in Signature Verification

Description

The `_verifySignature` function in `GdnRegistrationService` uses `abi.encodePacked` with dynamic types (strings) when creating the message hash. This can lead to hash collisions due to how `abi.encodePacked` concatenates parameters without padding, potentially allowing signature replay attacks with different input parameters.

Impact

- Malicious users could potentially forge valid signatures for unintended inputs
- Different combinations of `label` and `orderId` parameters could produce the same hash
- Could lead to unauthorized domain registrations or renewals

Fix

Replace `abi.encodePacked` with `abi.encode` to ensure proper parameter encoding:

```
bytes32 messageHash = keccak256(
    abi.encode(
        label,
        durationInYears,
        amountPaid,
        timestamp,
        userAddress,
        orderId
    )
);
```

[M-01] - Unbounded Loop in Domain Management Can Lead to DOS

Description

The `_removeStakerDomain` function in `GaiaDomainStaking.sol` contains an unbounded loop that iterates through all domains owned by a staker. This function is called when a staker's balance for a domain reaches zero, either through withdrawal or stake transfer. A malicious user could accumulate a large number of domains through multiple small stakes or transfers, making the loop too expensive to execute and effectively blocking stake withdrawals.

```
function _removeStakerDomain(address staker, string memory gdn) internal {
    string[] storage stakedGdns = stakerDomains[staker];
    for (uint i = 0; i < stakedGdns.length; i++) { // @audit: Unbounded
loop
```

```
            if (keccak256(bytes(stakedGdns[i])) == keccak256(bytes(gdn))) {
                stakedGdns[i] = stakedGdns[stakedGdns.length - 1];
                stakedGdns.pop();
                break;
            }
        }
    }
```

Impact

- High severity as it can permanently lock user funds
- Users with many domains could be unable to withdraw their stakes due to gas limits
- Malicious users could intentionally create this situation to grief other users
- The issue is compounded by the `transferStake` function which allows accumulating domains

Proof of Concept

1. Attacker stakes small amounts (e.g., 1 token) across many different domains (hundreds or thousands)
2. When attempting to withdraw or transfer stakes, the `_removeStakerDomain` function must iterate through all domains
3. If the number of domains is large enough, the gas cost will exceed block gas limits
4. The staker's funds become effectively locked as withdrawals will always revert

Fix

Here are some potential fixes:

1. Implement a maximum limit on domains per staker:

```
uint256 public constant MAX_DOMAINS_PER_STAKER = 100;

function stake(string calldata orderId, string calldata gdn, uint256
amount, bytes calldata signature) external {
    // ... existing checks ...

    if (stakeInfo.amount == 0) {
        require(stakerDomains[msg.sender].length < MAX_DOMAINS_PER_STAKER,
"Too many domains");
        stakeInfo.gdn = gdn;
        stakerDomains[msg.sender].push(gdn);
    }
}
```

2. Use a mapping instead of an array to track domains:

```
mapping(address => mapping(string => bool)) public stakerHasDomain;

function _removeStakerDomain(address staker, string memory gdn) internal {
    stakerHasDomain[staker][gdn] = false;
}
```

Our recommendation is to implement option 1 (maximum domain limit) combined with option 2 (mapping-based tracking) for the most robust solution. This provides clear bounds on gas usage while maintaining efficient domain management.

MEDIUM

[M-02] - Unsafe ERC721 Minting in GDN Domain System

Description

The GdnRegistrar contract uses _mint instead of _safeMint for NFT minting operations. This could lead to tokens being permanently locked if minted to contracts that don't support ERC721 token reception.

Impact

- Domain NFTs could become permanently locked if minted to incompatible contracts
- Loss of domain functionality and ownership
- Potential financial loss for users

Fix

```
function mint(
    string memory label,
    address to,
    uint256 expiry
) external onlyRegistrationService {
    // Checks
    require(
        registry.isLabelAvailable(label),
        "Label is already registered or not available"
    );
    require(to != address(0), "Cannot mint to zero address");

    // Effects
    uint256 tokenId = TokenIdCodec.encode(label, expiry);
    registry.setLabel(label, to, expiry);

    // Interactions
    _safeMint(to, tokenId);
    emit NftAction(1, label, to, expiry);
}

function renew(
    string memory label,
    uint256 duration
) external onlyRegistrationService {
    // Checks
    require(!isExpired(label), "Label is expired");
    (address owner, uint256 currentExpiry) = registry.getLabelInfo(label);
    require(owner != address(0), "Label not registered");

    // Effects
    uint256 newExpiry = currentExpiry + duration;
    uint256 tokenId = TokenIdCodec.encode(label, newExpiry);
    registry.setLabel(label, owner, newExpiry);

    // Interactions
    _safeMint(owner, tokenId);
    emit NftAction(2, label, owner, newExpiry);
}
```

[M-03] - `pausable` function needs to be implemented

Description

When inheriting from OpenZeppelin's Pausable contract, the internal _pause() and _unpause() functions are not automatically exposed. The contract needs to implement public/external functions that call these internal functions to make the pause functionality actually usable.

Impact

- The contract is not fully functional as it cannot be paused
- Users and other contracts relying on the pause functionality will not be able to use it
- Potential issues if the contract is not paused in critical situations

Fix

```
contract MyContract is Pausable, Ownable {
    function pause() external onlyOwner {
        _pause();
    }

    function unpause() external onlyOwner {
        _unpause();
    }
}
```

[M-04] - Set Beneficiary Should be a Two Step Process

Description

The setBeneficiary function updates the beneficiary address in a single step. If the new address is incorrectly specified, funds could be permanently lost. A two-step transfer pattern should be implemented to safely handle beneficiary changes.

Impact

- Funds could be permanently lost if the new address is incorrectly specified
- Users might not expect partial withdrawals to reset the unstaking state

Fix

```
address public pendingBeneficiary;

event BeneficiaryTransferStarted(address indexed currentBeneficiary,
address indexed pendingBeneficiary);
event BeneficiaryTransferCompleted(address indexed oldBeneficiary, address
indexed newBeneficiary);

function transferBeneficiary(address newBeneficiary) external onlyOwner {
    require(newBeneficiary != address(0), "Invalid address");
    require(newBeneficiary != beneficiary, "Already the beneficiary");

    pendingBeneficiary = newBeneficiary;
    emit BeneficiaryTransferStarted(beneficiary, newBeneficiary);
}

function acceptBeneficiaryTransfer() external {
    require(msg.sender == pendingBeneficiary, "Only pending beneficiary");
    require(pendingBeneficiary != address(0), "No pending transfer");

    address oldBeneficiary = beneficiary;
    beneficiary = pendingBeneficiary;
    pendingBeneficiary = address(0);

    emit BeneficiaryTransferCompleted(oldBeneficiary, beneficiary);
}
```

[M-05] - Unsafe Token Transfer Pattern in Reward Claims

Description

The `claimReward` function uses a direct `transfer` call to send GAIA tokens to the recipient. This can fail silently if the recipient is a smart contract that doesn't properly implement token reception or has complex logic in its receive function. Smart contracts may be unable to claim their rewards, leading to locked funds.

Impact

- Smart contracts may be unable to claim their earned rewards
- Potential permanent loss of rewards if recipient contract doesn't support token reception
- Poor interoperability with other DeFi protocols that might interact with the reward system

Fix

```
function claimReward() external {
    uint256 claimable = rewards[msg.sender].totalReward;
    require(claimable > 0, "No rewards to claim");

    // Clear state before transfer
    rewards[msg.sender].totalReward = 0;
    rewards[msg.sender].lastClaimRound = currentRound;

    // Use safe transfer pattern
    bool success = gaiaToken.transfer(msg.sender, claimable);
    require(success, "Token transfer failed");

    emit RewardClaimed(msg.sender, claimable);
}
```

LOW

[L-01] - Missing Zero Amount Validation in Reward Distribution Functions

Description

The reward distribution functions (`distributeStakerRewards`, `distributeDomainRewards`, and `distributeNodeRewards`) don't validate if calculated rewards are zero before updating state and emitting events. This leads to unnecessary gas costs from state updates and event emissions for zero-value rewards, and creates misleading event logs that could complicate off-chain tracking.

Fix

```
function distributeStakerRewards(...) external onlyRole(DISTRIBUTOR_ROLE)
{
    // ... existing code ...

    for (uint256 i = 0; i < stakers.length; i++) {
        // ... existing calculations ...
        uint256 reward = (dailyReward * domainWeights[i] *
STAKER_PERCENTAGE * stakedRatios[i]) /
            (difficulty * BASIS_POINTS * BASIS_POINTS);

        // Skip zero rewards
        if (reward == 0) continue;

        rewards[stakers[i]].totalReward += reward;
        rounds[currentRound].totalReward += reward;
        emit StakerRewardAdded(stakers[i], currentRound, orderIds[i],
reward);
    }
}
```

[L-02] - Indexed String Events Emit Hashes Instead of Values

Description

In the contract's events, strings marked as `indexed` (like in `LabelRegistered`) are stored as keccak256 hashes rather than the actual string values. This is evident in the test output where the indexed label appears as a hash (`0x6fd43e7...`) while the non-indexed version shows the actual string value ("example").

```
event LabelRegistered(
    string indexed label, // Stores hash: 0x6fd43e7c...
    address indexed owner,
    uint256 expiry
);
```

Impact

- Reduced event log usability for off-chain services
- More complex event filtering as services need to pre-compute hashes to filter by label
- No direct way to retrieve the original label string from indexed event logs

Fix

Remove the `indexed` keyword from string parameters in events:

```
event LabelRegistered(
    string label, // Remove indexed
    address indexed owner,
    uint256 expiry
);
```

[L-03] - Unchecked Array Length Matching in Distribution Functions

Description

The reward distribution functions (`distributeStakerRewards`, `distributeDomainRewards`, and `distributeNodeRewards`) accept multiple arrays as parameters but don't verify that these arrays have matching lengths. This could lead to out-of-bounds access or silent failures if arrays of different lengths are provided.

Impact

- Potential array index out of bounds errors
- Silent failures if arrays have mismatched lengths
- Possible partial distribution of rewards if arrays are mismatched

Fix

```
function distributeStakerRewards(
    address[] calldata stakers,
    uint256[] calldata domainWeights,
    uint256[] calldata stakedRatios,
    string[] calldata orderIds
) external onlyRole(DISTRIBUTOR_ROLE) {
    require(stakers.length <= MAX_BATCH_SIZE, "Batch too large");
+     require(
+         stakers.length == domainWeights.length &&
+         stakers.length == stakedRatios.length &&
+         stakers.length == orderIds.length,
+         "Array length mismatch"
+     );

    // ... rest of the function
}
```

INFORMATIONAL

[I-01] - Timestamp Checks Order: The timestamp checks are performed after creating the message hash.

Description

While this doesn't create an immediate vulnerability, it's better practice to validate inputs before performing operations with them. Here's how it could be reordered:

```
function _verifySignature(
    string memory label,
    uint256 durationInYears,
    uint256 amountPaid,
    uint256 timestamp,
    address userAddress,
    string memory orderId,
    bytes memory signature
) internal view returns (bool) {
    // Validate timestamp first
    require(timestamp <= block.timestamp, "Invalid timestamp");
    require(block.timestamp - timestamp < 15 minutes, "Expired
signature");

    bytes32 messageHash = keccak256(
        abi.encodePacked(
            label,
            durationInYears,
            amountPaid,
            timestamp,
            userAddress,
            orderId
        )
    );
    bytes32 ethSignedMessageHash = _toEthSignedMessageHash(messageHash);
    return ECDSA.recover(ethSignedMessageHash, signature) == signer;
}
```