Agora
NEAR-merkle-claim Audit

12-01-2025

COMPREHENSIVE REPORT

Prepared by:
Valhalla Security Consulting LLC

DISTRIBUTION (Public Version)

This report is intended for general informational use and provides an overview of the security testing performed by Valhalla Security Consulting LLC (Valhalla Security). All proprietary or sensitive details from the original assessment have been removed to ensure this document can be shared publicly.
The testing approach followed industry-standard penetration testing practices, adapted by Valhalla Security for this engagement. This public version may be freely distributed, referenced, or included in external communications as needed.

CONFIDENTIALITY (Public Version)

This public report has been prepared to summarize the high-level findings of the security assessment in a manner suitable for external audiences. All confidential, sensitive, or identifying information has been excluded.
Valhalla Security maintains strict confidentiality regarding all client security data. More detailed technical results, artifacts, or sensitive findings are contained only in the private version of this report and are shared exclusively with authorized client stakeholders.

ASSESSMENT TEAM

The following Valhalla Security personnel are involved in this engagement. Contact the appropriate personnel on this team to discuss the contents of this document or the work performed during this engagement.

PRIMARY SECURITY CONSULTANT

Ron S

Sr. Code
Auditor

ADDITIONAL SECURITY CONSULTANTS

Sean M
CEO – Valhalla Security

Alex B
CTO – Valhalla Security

Agora - NEAR-Merkle-claim Audit

Executive Summary

Agora enlisted the services of Valhalla Security to perform a time-boxed code audit assessment of the organization's NEAR-merkle-claim contracts. The engagement started on October 26th, 2025 for a one week review, and ended on November 3rd, 2025.

A phased approach to the code audit included information gathering, manual and automated testing, source code review, and adherence to best practices.

Valhalla Security identified several threats presenting risks of compromise and downstream risks. All findings were remediated, verified, and thoroughly tested.

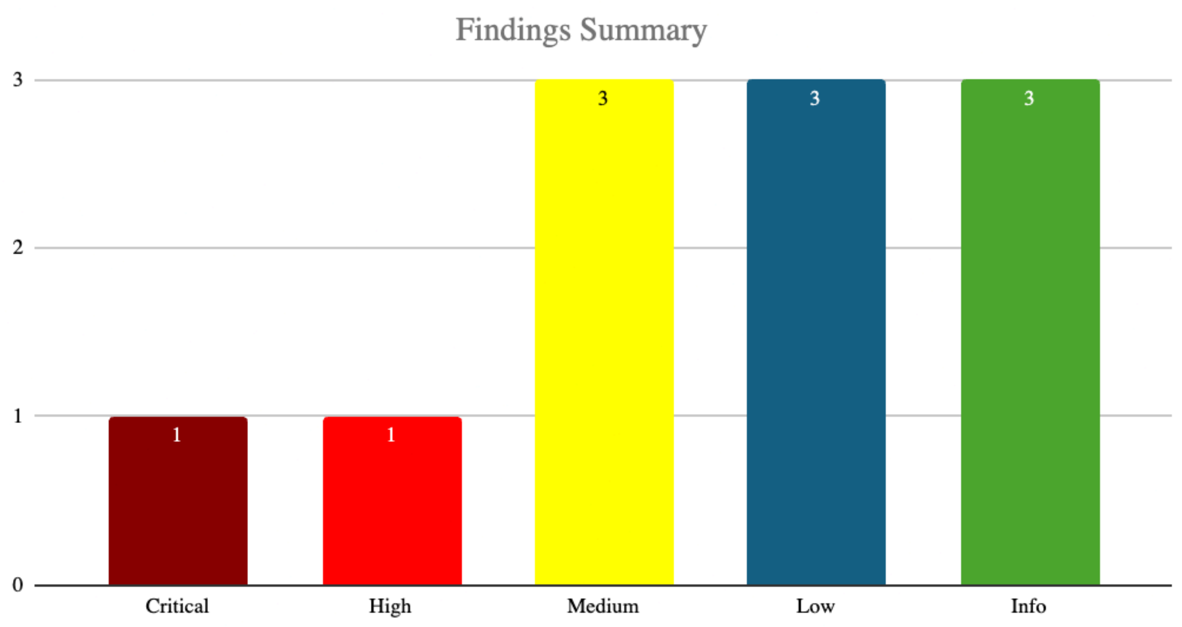
Scoped Repositories and Commits

NEAR-merkle-claim contract
Repository: https://github.com/voteagora/near-merkle-claim
Commit hash: 40442f9e0f5597e977f145bd4b8a62cce63ec55f

Areas of Focus

- Upgrade mechanisms
- Chain operations
- Claim delegation and operations
- Proof verification
- Fund handling
- Access controls
- Arithmetic operations
- Input validation
- Owner delegation and access

Finding Summary



Opportunities for improvements are identified below:

Critical Risk
1 - Inverted Claim Check Logic Allows Unlimited Double-Claims / Prevents Initial Claim - FIXED
High Risk
2 - Withdraw Function Drains Entire Balance Including Storage Reserve - FIXED
Medium Risk
3 - Campaign ID Overflow Risk - FIXED
4 - Missing Events Emitted for Critical Operations - FIXED
5 - Missing View Functions for Critical State - FIXED
Low Risk
6 - No Emergency Pause Mechanism - FIXED
7 - Test Coverage Issues - FIXED
8 - No Callback Handling for Transfer Failures - ACKNOWLEDGED / WILL NOT FIX
Informational
9 - Github Code Signing Not Enforced - ACKNOWLEDGED
10 - Missing Pull Request and Merge Strategy - ACKNOWLEDGED
11 - Vulnerable Rust Crate(s) - ACKNOWLEDGED

Positive Findings

In addition to identifying areas for improvement, this security audit recognizes several strengths in the design and implementation. These positive findings demonstrate a foundational level to security, contributing to a robust posture. Below, we highlight key aspects where the system is enforcing positive security habits.

- **The cryptographic implementation is sound**
- **Security fundamentals with access control and validation are in place**
- **The Merkle proof verification logic itself (src/merkle.rs) is implemented correctly using a commutative hashing and validation approach**
- **The architecture and design support real world use case**

These positive attributes reflect a solid foundation in secure coding practices and should be maintained and built upon in future development cycles and revisions. They collectively contribute to a more resilient product, helping to protect against common threats and supporting long-term security objectives.

Critical Risk

1 - Inverted Claim Check Logic Allows Unlimited Double-Claims / Prevents Initial Claim	
Severity	**CRITICAL**
Location	**src/lib.rs:108-111**
Status	**Fixed / Remediated**
Description	The logic in the claim check is inverted. The condition checks *if != None* (user has claimed), when it should check == *None* (user has not claimed). `require!(` `self.claims.get(&campaign_id) != None,` `"Already claimed rewards"` `);`
Impact	This check indicates that users who have already claimed can claim again infinitely, and users who have not claimed cannot claim at all. This results in the contract being broken and unusable.
Proposed Fix	Update the check logic to use "== None" or the idiomatic method listed below. This ensures the logic is functioning correctly. `require!(` `self.claims.get(&campaign_id) == None,`

1 - Inverted Claim Check Logic Allows Unlimited Double-Claims / Prevents Initial Claim	
	```
	    "Already claimed rewards"
	);

	Or (idiomatically):

	require!(
	    !self.claims.contains_key(&campaign_id),
	    "Already claimed rewards"
	);
	```
Status Notes	Fixed in https://github.com/voteagora/near-merkle-claim/commit/23f888d157380f9d99dee89d20e7b989d782b3fc

High Risk

2 - Withdraw Function Drains Entire Balance Including Storage Reserve	
Severity	**HIGH**
Location	**src/lib.rs:151**
Status	**Fixed / Remediated**
Description	The code for the withdraw function calls a transfer using the entire *account_balance(),* without reserving funds for storage state costs. `Promise::new(caller).transfer(env::account_balance());`
Impact	The contract may not have enough balance to cover storage and this could cause the contract to become non-functional. Additionally, this violates NEAR storage staking requirements and the contract may panic on subsequent operations requiring storage. Note: If the contract is designed to never be used again or will be deleted and cleaned up after funds are removed, this finding would move to an Informational status, rather than High. If the contracts are designed to be used again in the future, or interacted with in any way that affects storage (*self.campaigns* and *self.claims)* then it is recommended to leave a small balance (0.1 NEAR) or estimate storage costs. Reference: https://docs.near.org/concepts/storage/storage-staking

2 - Withdraw Function Drains Entire Balance Including Storage Reserve

Proposed Fix

Calculate the minimum balance needed for storage per contract or use a static amount if known costs will be less (0.1 NEAR for example).

```
pub fn withdraw(&mut self) {
    let caller = env::predecessor_account_id();
    require!(
        caller == self.config.owner_account_id,
        "Caller must be the owner of the claims contract"
    );
    // Calculate minimum balance needed for storage
    let storage_cost = env::storage_byte_cost()
        .saturating_mul(env::storage_usage() as u128);
    let available_balance = env::account_balance()
        .saturating_sub(storage_cost);
    require!(
        available_balance > 0,
        "No available balance to withdraw"
    );
    Promise::new(caller).transfer(NearToken::from_yoctonear(available_balance));
}
```

Status Notes

Fixed in:

https://github.com/voteagora/near-merkle-claim/commit/c6f1f336af28f4fa4e4cd30fedd584706b77075b - Add a min storage deposit that cannot be withdrawn

https://github.com/voteagora/near-merkle-claim/commit/e771ece2aa55069c3c96a9e93dc92d4b9836e5d1 - Move logic to constructor

https://github.com/voteagora/near-merkle-claim/commit/72df198e09ab33cd43075bde50022356874150fe - Make new payable

https://github.com/voteagora/near-merkle-claim/commit/3867cb81e5372c5ef28fcaef4ae273fde7fd27a7 - Fix storage costs and self fund transfer

Medium Risk

3 - Campaign ID Overflow Risk	
Severity	**MEDIUM**
Location	**src/lib.rs:81-91**

3 - Campaign ID Overflow Risk	
Status	**Fixed / Remediated**
Description	If *last_campaign_id* reaches u32::MAX (4,294,967,295), the addition will overflow.
Impact	Contract would panic when creating campaign #4,294,967,296 resulting in no more campaigns being created (contract permanently bricked for this function). While unlikely in practice, it's a permanent DoS vector and should be remediated.
Proposed Fix	Perform a < u32::MAX check indicating that a max number of campaigns have been reached.

```rust
pub fn create_campaign(&mut self, merkle_root: CryptoHash, claim_end: U64) {
    require!(
        env::predecessor_account_id() == self.config.owner_account_id,
        "Account must be the owner"
    );
    require!(
        env::block_timestamp() < claim_end.into(),
        "Claim end timestamp must be some time in the future"
    );
    // Check for overflow before incrementing
    require!(
        self.last_campaign_id < u32::MAX,
        "Maximum number of campaigns reached"
    );
    self.last_campaign_id = self.last_campaign_id
        .checked_add(1)
        .expect("Campaign ID overflow");
    let campaign = RewardCampaign {
        id: self.last_campaign_id,
        claim_start: env::block_timestamp().into(),
        claim_end,
        merkle_root,
    };
    self.campaigns.insert(self.last_campaign_id, campaign.into());
}
```

Status Notes	Fixed in https://github.com/voteagora/near-merkle-claim/commit/2c1ede708f606ca56cda97927396efc075410b38

4 - Missing Events Emitted for Critical Operations	
Severity	**MEDIUM**
Location	**src/lib.rs (multiple)**
Status	**Fixed / Remediated**
Description	Events are not being emitted for operations such as: - Campaign creation - Claims - Withdrawals
Impact	This can cause issues with auditing on-chain activities, audit trails, and introduces difficulty when tracking contract activity. Additionally, this can make it more difficult when building future indexers or frontends for contracts and contract activity.

Proposed Fix	Use the near_sdk:log function to add event logging macros and event emission.
	
	`use near_sdk::log;`
	`// Add event logging macros`
	`#[near]`
	`pub struct CampaignCreatedEvent {`
	`    pub campaign_id: CampaignId,`
	`    pub merkle_root: CryptoHash,`
	`    pub claim_end: U64,`
	`}`
	`#[near]`
	`pub struct ClaimEvent {`
	`    pub campaign_id: CampaignId,`
	`    pub account_id: AccountId,`
	`    pub lockup_contract: AccountId,`
	`    pub amount: Balance,`
	`}`
	`// In create_campaign:`
	`log!("EVENT_JSON:{}", serde_json::to_string(&CampaignCreatedEvent { ... }).unwrap());`
	`// In claim:`
	`log!("EVENT_JSON:{}", serde_json::to_string(&ClaimEvent { ... }).unwrap());`

Status Notes	Fixed in https://github.com/voteagora/near-merkle-claim/commit/3a7d422c7b566f7a09a64ae38b7141a41399c396

5 - Missing View Functions for Critical State	
Severity	**MEDIUM**
Location	**src/lib.rs**
Status	**Fixed / Remediated**
Description	There are no view functions to query: - Campaign details by ID - Whether an account has claimed for a campaign - List of all campaigns - Claim status for a specific account/campaign pair
Impact	Without a view query function, users cannot verify if they've claimed already without making a subsequent transaction and frontend/tooling cannot easily query contract state. This makes it more difficult to debug issues and can create a poor user experience.

Proposed Fix	Add various view functions to provide end-users with more information.
	
	`// Add these view functions`
	`pub fn get_campaign(&self, campaign_id: CampaignId) -> Option<RewardCampaign> {`
	`    self.campaigns.get(&campaign_id).map(|c| c.clone())`
	`}`
	`pub fn has_claimed(&self, campaign_id: CampaignId, account_id: AccountId) -> bool {`
	`    self.claims.get(&campaign_id)`
	`        .map(|claimed_account| claimed_account == account_id)`
	`        .unwrap_or(false)`
	`}`
	`pub fn get_last_campaign_id(&self) -> CampaignId {`
	`    self.last_campaign_id`
	`}`

Status Notes	Fixed in https://github.com/voteagora/near-merkle-claim/commit/d0635e53fb1fb3911e93d9787b18c98b5bbd85d7

Low Risk

6 - No Emergency Pause Mechanism	
Severity	**LOW**
Location	**src/lib.rs**

6 - No Emergency Pause Mechanism	
Status	**Fixed / Remediated**
Description	There is currently no way to pause the contract in case of emergency.
Impact	If a vulnerability is discovered, the owner or user cannot stop claims immediately. The claim contract could lose more funds during an incident. Note: By allowing an owner account to pause a contract, a malicious owner could deny legitimate user claims and funds. If a pausing mechanism is introduced, it should be transparent via event emission/views/frontend and/or include the owner to use a multisignature signing system or key to call the pause/unpause functionality.
Proposed Fix	Add pause boolean, methods and updates into the *claim()* function for contract pausing.

```
// Add to contract state
pub struct MerkleClaim {
    // ... existing fields
    paused: bool,
}
// Add pause/unpause methods
pub fn pause(&mut self) {
    require!(
        env::predecessor_account_id() == self.config.owner_account_id,
        "Only owner can pause"
    );
    self.paused = true;
}
pub fn unpause(&mut self) {
    require!(
        env::predecessor_account_id() == self.config.owner_account_id,
        "Only owner can unpause"
    );
    self.paused = false;
}
// Add check in claim function
pub fn claim(...) {
    require!(!self.paused, "Contract is paused");
    // ... rest of function
}
```

Status Notes	Fixed in https://github.com/voteagora/near-merkle-claim/pull/9/commits/6fe5b92dd5167523ce9e8130782fd8e53b55dc36

7 - Test Coverage Issues	
Severity	**LOW**
Location	**tests/test_basics.rs**
Status	**Fixed / Remediated**
Description	The test file and directory contains essentially no real tests or testing. Test code that does exist is commented out.
Impact	By not including robust testing and test cases, there is no validation of contract behavior, vulnerabilities may not be caught during development, and future breaking changes may not be detected.
Proposed Fix	Implement comprehensive tests including: - Normal “happy path” claim flow - Double-claim prevention - Campaign creation - Withdrawal after claim period expiration - Edge cases (invalid proofs, expired campaigns, wrong *account_id*, etc.)
Status Notes	Fixed in: https://github.com/voteagora/near-merkle-claim/commit/f8acb4c - Initialize contract context for testing https://github.com/voteagora/near-merkle-claim/commit/649865e4a465754bb2ce85d319f13479a6501340 - Test helpers for campaign https://github.com/voteagora/near-merkle-claim/commit/88757514edd588332b31ade9e15aa1dc655571ef - Add remaining unit tests

8 - No Callback Handling for Transfer Failures	
Severity	**HIGH**
Location	**src/lib.rs:139**
Status	**Acknowledged / Will Not Fix**
Description	The transfer to the lockup contract has no callback to verify success. If the transfer fails (lockup contract doesn't exist, is malformed, or rejects), the claim is still marked as complete.

8 - No Callback Handling for Transfer Failures	
	`Promise::new(lockup_contract).transfer(NearToken::from_yoctonear(amount.0));`
Impact	If a transfer to a lockup contract fails, the user could lose their claim permanently, funds may be stuck in contract but marked as claimed, and there is currently no way to recover or retry. Reference: https://docs.near.org/smart-contracts/anatomy/crosscontract

Proposed Fix	Add a callback method into the *claim()* function.
	
	`// Add a callback method`
	`#[private]`
	`pub fn claim_callback(`
	`    &mut self,`
	`    campaign_id: CampaignId,`
	`    account_id: AccountId,`
	`    #[callback_result] call_result: Result<(), near_sdk::PromiseError>,`
	`) {`
	`    if call_result.is_err() {`
	`        // Rollback the claim on failure`
	`        self.claims.remove(&campaign_id);`
	`        env::panic_str("Transfer failed, claim rolled back");`
	`    }`
	`}`
	`// In claim function:`
	`Promise::new(lockup_contract)`
	`    .transfer(NearToken::from_yoctonear(amount.0))`
	`    .then(`
	`        Self::ext(env::current_account_id())`
	`            .claim_callback(campaign_id, user_account_id.clone())`
	`    );`

Status Notes	The client acknowledges the security benefit of implementing callback handling as a defensive programming practice, but upon further analysis of the repository the risk severity has been reconsidered and the issue will not be fixed. The *claim()* function cryptographically validates the lockup_contract address through merkle proof verification before executing any transfer. This design ensures that lockup addresses cannot be arbitrarily provided by users and must be predetermined during merkle tree generation. Transfer failures would only occur in the following scenarios: - Administrative configuration errors during merkle tree construction

8 - No Callback Handling for Transfer Failures	
	- Non-existent account IDs included in the merkle tree - Race conditions where accounts are deleted between tree generation and claim execution The client has decided not to implement callback handling in the current version and the finding is being marked as Acknowledged, but will not be fixed due to the findings from further analysis.

Informational Risk

9 - Github Code Signing Not Enforced	
Severity	**Informational**
Repository	**near-merkle-claim**
Description	The audited GitHub repository does not require or enforce commit signing with GPG keys. This means there's no cryptographic verification to ensure commits come from trusted developers.
Impact	Not a direct vulnerability, but lack of code signing could make it easier for malicious code from a compromised developer account or laptop to be merged through unnoticed.
Proposed Fix	Enforce Code Signing on All Repositories Step 1: Access Branch Protection Settings Step 2: Configure Branch Protection to Require Signed Commits Step 3: Check **Require signed commits**

10 - Missing Pull Request and Merge Strategy	
Severity	**Informational**
Repository	**near-merkle-claim**

10 - Missing Pull Request and Merge Strategy	
Description	The audited GitHub repository allows development activity to be committed directly to the **main/master** branch without requiring pull requests (PRs). There’s no enforced process to ensure a separate QA/merge role reviews changes before they’re merged, which could allow unverified or malicious code to be pushed, especially if a developer’s account or laptop is compromised.
Impact	Malicious code risk if developer account or laptop is compromised.
Proposed Fix	Set Branch Protection Settings **Step 1: Access Branch Protection Settings** 1. Navigate to the repository 2. Click the **Settings** tab at the top. 3. In the left sidebar, under **Code and automation**, click **Branches**. 4. Under **Branch protection rules**, find the main or master branch or click **Add rule** to create a new one. **Step 2: Configure Branch Protection to Require Pull Requests** 1. In the branch protection rule settings, check **Protect this branch**. 2. Check **Require a pull request before merging** to block direct commits to main or master. 3. Check **Require approvals** and set the minimum number of reviewers (e.g., 1 or 2) to ensure a separate QA/merge role reviews the PR. 4. Check **Require signed commits** to ensure commits are cryptographically verified, further reducing risk from compromised accounts. **Step 3: Additional Protections and Verification** 1. Check **Dismiss stale pull request approvals when new commits are pushed** to ensure fresh reviews if code changes after initial approval.

11 - Vulnerable Rust Crate(s)	
Severity	**Informational**
Repository	**near-merkle-claim**
Description	The audited GitHub repository uses third-party packages and dependencies that may have known vulnerabilities due to outdated versions. These packages, such as libraries or frameworks, are not regularly updated to their latest secure versions, leaving the codebase exposed to potential exploits.

11 - Vulnerable Rust Crate(s)	
Impact	Failing to update vulnerable third-party dependencies can increase the risk of security breaches or exploits. *wee_alloc* specifically can cause memory leaks causing crashes over time.
Vulnerable Crates (Rust)	**Crate: wee_alloc** Version: 0.4.5 Warning: unmaintained Title: wee_alloc is Unmaintained Date: 2022-05-11 ID: RUSTSEC-2022-0054 URL: https://rustsec.org/advisories/RUSTSEC-2022-0054 Dependency tree: wee_alloc 0.4.5 └── near-sdk 5.17.1 └── near-merkle-claim 0.1.0 warning: 1 allowed warning found Reference(s): https://github.com/rustwasm/wee_alloc/issues/106 https://www.reddit.com/r/rust/comments/pyvsz3/wee_alloc_vs_dlmalloc_wasm_target/ https://www.reddit.com/r/rust/comments/x1cle0/dont_use_wee_alloc_in_production_code_targeting/
Proposed Fix	- Determine if *wee_alloc* is required and document if so - Migrate to a newer more updated package providing the same functionality as *wee_alloc* - Use an alternative allocator such as *dlmalloc* by configuring the default features in the near-sdk - Upgrade to near-sdk version 5.17.2

Tools

- anchor
- cargo-audit
- cargo-clippy (linter)
- cargo-fuzz (fuzzer)
- semgrep

List of References

- https://docs.near.org/smart-contracts/security/welcome

- https://docs.near.org/smart-contracts/anatomy/crosscontract
- https://github.com/rustwasm/wee_alloc/issues/106
- https://docs.near.org/concepts/storage/storage-staking
- https://www.reddit.com/r/rust/comments/pyvsz3/wee_alloc_vs_dlmalloc_wasm_target/
- https://www.reddit.com/r/rust/comments/x1cle0/dont_use_wee_alloc_in_production_code_targeting/