Guvenkaya®

The Sweat Foundation Ltd

SWEAT NEP-141 Token Smart Contract Security Review

Principal Advisor: Timur Güvenkaya
Date of Engagement: 29th May 2026 - 5th June 2026
Visit: www.guvenkaya.co

Contents

About Us

Security Reviews, Secure Design, and Advisory for High-Stakes Digital Systems.

Guvenkaya Advisory helps digital asset operators, financial institutions, and security-critical technology teams secure the systems and workflows behind digital value. Trusted across blockchain ecosystems and financial technology, we provide principal-led security reviews, penetration testing, secure design, custody and key-management review, technical due diligence, and digital asset advisory.

About The Sweat Foundation

The Sweat Foundation is the organization behind Sweat Economy, an application that sits at the intersection of fitness and crypto. It motivates users to stay active by converting steps into SWEAT Token and works as an accessible entry point into crypto for many users.

Audit Results

Guvenkaya conducted a security assessment of the SWEAT NEP-141 token contract on NEAR, including the deferred minting flow, storage management behavior, migration path, access control wiring, pause behavior, and Sweat-specific changes to the forked vendored fungible token implementation.

During this engagement, 11 findings were reported: 1 High, 1 Medium, 7 Low, and 2 Informational severity issues. The High finding concerns storage accounting undercharge risk in the custom **`LookupMapAdapter`**. The Medium finding concerns runtime governance coupling. The Low findings cover deferred minting hardening, pause controls, fee calculation, burn authorization hardening, and batch gas limits. The Informational findings cover missing event emissions for operational observability and an acknowledged deferred-batch overlap edge case.

Project Scope

SWEAT NEP-141 token contract

Files	Link
api.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/contract/src/api.rs
core.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/contract/src/core.rs
defer.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/contract/src/defer.rs
lib.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/contract/src/lib.rs
math.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/contract/src/math.rs
metadata.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/contract/src/metadata.rs
migration.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/contract/src/migration.rs
storage.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/contract/src/storage.rs

Project Scope (continued)

The vendored files below were reviewed only for the Sweat-specific modifications made in this fork, such as the custom **`LookupMapAdapter`**. The upstream common fungible token library was not reviewed in full.

Files	Link
core_impl.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/vendors/near-contract-standards/src/fungible_token/core_impl.rs
storage_impl.rs	https://github.com/Sweat-Foundation/sweat-token/blob/9b84e71ae05d7faf46826a82fbaf453794f0ef14/vendors/near-contract-standards/src/fungible_token/storage_impl.rs

Out of Scope

The audit focused on the code paths listed in scope and their smart contract security properties. It did not include a full review of off-chain services, oracle infrastructure beyond how oracle data is accepted by the contract, deployment operations, private keys, tests, or external dependencies. The upstream NEAR fungible token standard library was out of scope except for the Sweat-specific changes to the forked vendored files identified above.

Timeline

Start of the audit	Draft report	Final report
29th May 2026	5th June 2026	5th June 2026

Methodology

RESEARCH INTO PROJECT ARCHITECTURE

PREPARING ATTACK VECTORS

SETTING UP AN ENVIRONMENT

MANUAL CODE REVIEW OF THE CODE

ASSESSMENT OF RUST SECURITY ISSUES

ASSESSMENT OF NEAR SECURITY ISSUES

ASSESSMENT OF ARITHMETIC ISSUES

BUSINESS LOGIC VULNERABILITY ASSESSMENT

ONCHAIN TESTING USING NEAR WORKSPACES

BEST PRACTICES AND CODE QUALITY

CHECKING FOR CODE REFACTORING/SIMPLIFICATION POSSIBILITIES

ARCHITECTURE IMPROVEMENT SUGGESTIONS

PREPARING POCS AND/OR TESTS FOR EACH CRITICAL/HIGH/MEDIUM ISSUES

Severity Breakdown

Guvenkaya rates findings by likelihood and impact so remediation can be prioritized by technical risk, business context, and the decision the engagement is meant to support.

01. Likelihood Ratings

Likely: The vulnerability is easily discoverable and not overly complex to exploit.
Possible: The vulnerability presents some challenges either in discovery or in the complexity of the attack.
Rare: The vulnerability is either very difficult to discover or complex to exploit, or both.
This matrix provides a nuanced view, taking into account both the ease of discovering a vulnerability and the complexity involved in exploiting it.

02. Impact

Severe: Exploitation could result in critical loss or compromise, such as full system control, substantial financial loss, or severe reputational damage.
High: Exploitation could materially affect governance, accounting, availability, or user funds, but does not imply full system compromise under the assessed conditions.
Moderate: Exploitation may lead to limited data loss, partial compromise, moderate financial impact, or noticeable degradation of services.
Negligible: Exploitation has minimal impact, such as minor data exposure without significant consequences or slight inconvenience without substantial disruption.

03. Severity Ratings

Critical: Assigned to vulnerabilities with severe impact and a likely likelihood of exploitation.
High: For vulnerabilities with either severe impact but only a possible likelihood, or high impact with a likely or possible likelihood.
Medium: Used for vulnerabilities with severe impact but a rare likelihood, high impact with a rare likelihood, moderate impact with a possible likelihood, or negligible impact with a likely likelihood.
Low: For vulnerabilities with moderate impact and rare likelihood, or negligible impact with a possible likelihood.
Informational: The lowest severity rating, typically for vulnerabilities with negligible impact and a rare likelihood of exploitation.

CRITICAL **HIGH** **MEDIUM** Low Informational

Likelihood Matrix:

Attack Complexity \ Discovery Ease	Obvious	Concealed	Hidden
Complex	Possible	Rare	Rare
Moderate	Likely	Possible	Rare
Straightforward	Likely	Possible	Possible

Likelihood/Impact Matrix:

Likelihood \ Impact	Severe	High	Moderate	Negligible
Likely	**CRITICAL**	**HIGH**	**HIGH**	**MEDIUM**
Possible	**HIGH**	**HIGH**	**MEDIUM**	Low
Rare	**MEDIUM**	**MEDIUM**	Low	Informational

Findings Summary

01. Remediation Complexity: This measures how difficult it is to remediate a finding once it has been identified.

Simple: Patches or fixes are readily available and easily implemented.

Moderate: Requires some time and resources to remediate, but well within the capabilities of most organizations.

Difficult: Remediation requires significant resources, specialized skills, or substantial changes to systems or architecture.

02. Status: This tracks the client's response to each finding and whether remediation has been completed, accepted, or scheduled.

Not Fixed: Indicates that the vulnerability has been identified but no remedial action has been taken yet. This status is crucial for newly discovered vulnerabilities or those awaiting prioritization.

Fixed: This status is applied when the vulnerability has been successfully remediated. It implies that appropriate measures, such as patching, configuration changes, or architectural modifications, have been implemented to resolve the issue.

Acknowledged: This status is used for vulnerabilities that have been recognized, but for various reasons (such as risk acceptance, cost, or other business decisions), have not been fixed. It indicates that the risk posed by the vulnerability is known and has been consciously accepted.

Scheduled: This status indicates that the vulnerability has been acknowledged and a plan is in place to fix it in the future. It signifies that while remediation hasn't yet occurred, the issue has been prioritized and is part of the planned development roadmap.

Finding	Impact	Likelihood	Severity	Remediation Complexity	Remediation Status
GUV-1: LookupMap adapter can undercharge storage for selected accounts	High	Possible	HIGH	Moderate	Fixed
GUV-2: Runtime governance was coupled to contract self-call authority	High	Rare	MEDIUM	Moderate	Fixed
GUV-3: Denylisted users could still receive deferred mints	Moderate	Rare	Low	Simple	Fixed
GUV-4: Fee calculation could overflow before division	Moderate	Rare	Low	Simple	Fixed
GUV-5: burn did not require one yoctoNEAR	Moderate	Rare	Low	Simple	Fixed
GUV-6: Forced storage_unregister was not covered by token pause	Moderate	Rare	Low	Simple	Fixed
GUV-7: Arbitrary pause keys can bloat storage through third-party pausable dependency	Moderate	Rare	Low	Moderate	Fixed
GUV-8: Deferred minting accepted empty and zero-step batches	Moderate	Rare	Low	Simple	Fixed
GUV-9: Deferred minting had no explicit max batch size	Moderate	Rare	Low	Simple	Fixed
GUV-10: Missing event emissions reduced auditability	Negligible	Rare	Informational	Simple	Fixed
GUV-11: Overlapping deferred batches can reuse step ranges after rollback	Negligible	Rare	Informational	Simple	Acknowledged

Findings Details

GUV-1 LookupMap adapter can undercharge storage for selected accounts - High

The vendored fungible token implementation used a custom **`LookupMapAdapter`** that could store either **`sha256(AccountId)`** keys or unhashed account ID keys depending on a configured postfix. Accounts matching the unhashed path could consume different storage than the measured 64-byte probe account used for **`account_storage_usage`**.

Because NEP-145 storage deposits are charged from **`account_storage_usage`**, selected account names could be undercharged and storage costs could depend on account naming rather than a single stable key format.

Affected area: LookupMapAdapter

```
pub struct LookupMapAdapter {
   inner: LookupMap<LookupMapKey, Balance>,
   skip_hashing_postfix: Option<String>,
}
```

Recommendation

Use a single account-key encoding for all users and measure storage against that exact encoding. Preserve legacy trie keys during migration so existing balances remain addressable.

Remediation - Fixed

The adapter now stores balances under the hashed key variant, while migration reads the old layout and reconstructs the token with **`from_prefix`**. Fixed in commit **afd398e6e092ab4fbff6d14162b9d0529bc2c6cf**.

Fixed code: hashed key variant

```
pub enum LookupMapKey {
  Hash([u8; 32]),
}

fn hash_key(account: &AccountId) -> LookupMapKey {
  LookupMapKey::Hash(env::sha256_array(account.as_bytes()))
}
```

GUV-2 Runtime governance was coupled to contract self-call authority - Medium

The initial ACL bootstrap treated the contract account as the super-admin by default. This coupled runtime governance with contract self-call or upgrade authority and made it harder to assign operational roles to the intended accounts during initialization or migration.

The issue is medium severity because compromised or misconfigured governance authority can have high operational impact, while exploitation remains rare due to the required authority path.

Affected area: initializer authority

```
#[init]
fn new(holding_account_id: AccountId) -> Self {
    let mut contract = Self { /* ... */ };
    contract.acl_init_super_admin(env::current_account_id());
    contract
}
```

Recommendation

Accept explicit governance accounts during initialization and migration, and initialize ACL roles through an internal unchecked grant path only during trusted setup flows.

Remediation - Fixed

The initializer and migration now accept explicit super-admin, oracle, denylist manager, pause manager, and unpause manager accounts. Fixed in commit **779e31c2951209c5c7777de160d0b937f9823775**.

Fixed code: explicit ACL bootstrap

```
fn init_acl(
    &mut self,
    super_admin_account_id: AccountId,
    oracle_account_ids: Vec<AccountId>,
    denylist_manager_account_ids: Vec<AccountId>,
    pause_manager_account_ids: Vec<AccountId>,
    unpause_manager_account_ids: Vec<AccountId>,
) {
    self.acl_init_super_admin(super_admin_account_id);

    for (role, account_ids) in [
      (Role::Oracle, oracle_account_ids),
      (Role::DenylistManager, denylist_manager_account_ids),
      (Role::PauseManager, pause_manager_account_ids),
      (Role::UnpauseManager, unpause_manager_account_ids),
    ] {
      for account_id in account_ids {
        self.acl_get_or_init().grant_role_unchecked(role, &account_id);
      }
    }
}
```

GUV-3 Denylisted users could still receive deferred mints - Low

The deferred minting path computed rewards for every **`(account_id, step_count)`** entry in **`defer_batch`** and sent each user amount to the holding contract. A denylisted account could therefore still be credited through the deferred claim ledger even though direct transfer and burn paths rejected restricted users.

Affected area: defer_batch loop

```
for (account_id, step_count) in steps_batch {
  let (amount, fee) = self.calculate_tokens_amount(step_count);
  self.steps_since_tge.0 += u64::from(step_count);
  accounts_tokens.push((account_id, U128(amount)));
}
```

Recommendation

Check each batch recipient against the denylist before calculating or recording any mint amount for that recipient.

Remediation - Fixed

The loop now skips restricted accounts before payout calculation, step accounting, or holding-contract payload construction. Fixed in commit **d51de9307ac4555ba87fa53d9b97d3cbd27d8b56**.

Fixed code: denylist skip before accounting

```
for (account_id, step_count) in steps_batch {
  if self.is_restricted(&account_id) {
    continue;
  }

  let (amount, fee) = self.calculate_tokens_amount(step_count);
  self.steps_since_tge.0 += u64::from(step_count);
  accounts_tokens.push((account_id, U128(amount)));
}
```

GUV-4 Fee calculation could overflow before division - Low

The payout split calculated the oracle fee as **`(value * 5).div_ceil(100)`**. For sufficiently large **`u128`** input, the multiplication could overflow before the division reduced the value.

The issuance formula is not expected to produce values near **`u128::MAX`** under normal operation, but fee calculation is consensus relevant and should avoid unnecessary overflow patterns.

Affected area: fee calculation

```
let fee = (value * 5).div_ceil(100);
```

Recommendation

Rewrite the fee calculation into an equivalent expression that avoids multiplication before division.

Remediation - Fixed

The fee calculation was rewritten to **`value.div_ceil(20)`**, which is equivalent to a 5 percent rounded-up fee and avoids intermediate multiplication overflow. Fixed in commit **d6d481cf5de26b52143dd4ed9d79e77cba24ba69**.

Fixed code: overflow-safe fee calculation

```
let fee = value.div_ceil(20); // == value * 5 / 100
```

GUV-5 burn did not require one yoctoNEAR - Low

The custom **`burn`** method mutates token balances but did not require exactly one yoctoNEAR. NEAR token-standard mutating calls typically require one yoctoNEAR to prevent accidental or restricted access-key invocation.

Because **`burn`** destroys user funds, it should follow the same intent-confirmation pattern used by **`ft_transfer`**, **`ft_transfer_call`**, and storage unregister.

Affected area: burn

```
fn burn(&mut self, amount: U128) {
   self.assert_not_in_denylist(vec![&env::predecessor_account_id()]);
   self.token.internal_withdraw(&env::predecessor_account_id(), amount.0);
}
```

Recommendation

Mark **`burn`** as payable and require **`assert_one_yocto()`** before withdrawing the caller balance.

Remediation - Fixed

The method is now **`#[payable]`** and calls **`assert_one_yocto()`** before denylist checks and balance withdrawal. Fixed in commit **ce514f6d533ce35a63a5b2a539052057567d290f**.

Fixed code: payable burn with one yoctoNEAR

```
#[payable]
fn burn(&mut self, amount: U128) {
   self.assert_feature_enabled(Feature::Token);
   assert_one_yocto();
   self.assert_not_in_denylist(vec![&env::predecessor_account_id()]);

   self.token.internal_withdraw(&env::predecessor_account_id(), amount.0);
}
```

GUV-6 Forced storage_unregister was not covered by token pause - Low

`storage_unregister(force = true)` can burn the caller's token balance while removing their storage registration. Before remediation, token transfers and burns were covered by the **`token`** pause feature, but forced unregister was not.

During a token pause, users could still execute a state-mutating path that burns balances and changes storage registration.

Affected area: storage_unregister

```
#[payable]
fn storage_unregister(&mut self, force: Option<bool>) -> bool {
   self.assert_not_in_denylist(vec![&env::predecessor_account_id()]);
   self.token.internal_storage_unregister(force).is_some()
}
```

Recommendation

Apply the token pause guard to **`storage_unregister`** so the pause surface covers all user-facing token balance mutation paths.

Remediation - Fixed

`storage_unregister` now calls **`assert_feature_enabled(Feature::Token)`**, which blocks forced unregister while the token feature is paused. Fixed in commit **319cb9408423a231069bf7fd300cd5b29c9e31d9**.

Fixed code: token pause guard

```
#[payable]
fn storage_unregister(&mut self, force: Option<bool>) -> bool {
  self.assert_feature_enabled(Feature::Token);
  self.assert_not_in_denylist(vec![&env::predecessor_account_id()]);

  self.token.internal_storage_unregister(force).is_some()
}
```

GUV-7 Arbitrary pause keys can bloat storage through third-party pausable dependency - Low

The third-party **`near-plugins`** pausable implementation stores paused feature keys as arbitrary strings. If a **`PauseManager`** role is compromised, the attacker can repeatedly pause large or junk keys and force the contract to persist them in the pause-key set.

The contract only needs a small fixed pause domain, such as token operations, minting, and all features. Allowing arbitrary pause-key storage increases dependency risk and gives a compromised role an avoidable storage-bloat primitive.

Affected dependency behavior

```
fn pa_pause_feature(&mut self, key: String) -> bool {
    let mut paused_keys = self.pa_all_paused().unwrap_or_default();
    let newly_paused = paused_keys.insert(key.clone());
    env::storage_write(
        self.pa_storage_key().as_ref(),
        borsh::to_vec(&paused_keys).unwrap().as_ref(),
    );
    true
}
```

Recommendation

Replace arbitrary-string pause storage with a small internal bitmap and reject unknown pause features.

Remediation - Fixed

The remediation replaces arbitrary-key pause storage with fixed feature bits for token and minting operations while preserving the pause-manager and unpause-manager access model. Pause state is now stored as a **`u32`** bitmask on contract state, and public pause calls accept only the **`Feature`** enum values. Fixed in commit **3a6a0bf77f34f4ed9a56d660fb98ee2c4f40d0bd**.

Fixed code: feature bitmask pause state

```
pub enum Feature {
    Token,
    Minting,
}

impl Feature {
    const fn bit(self) -> u32 {
        match self {
            Feature::Token => 1 << 0,
            Feature::Minting => 1 << 1,
        }
    }
}

fn set_features_paused(&mut self, features: Vec<Feature>, paused: bool) -> bool {
    let before = self.paused_features;

    for feature in features {
        if paused {
            self.paused_features |= feature.bit();
        } else {
            self.paused_features &= !feature.bit();
        }
    }

    self.paused_features != before
}
```

GUV-8 Deferred minting accepted empty and zero-step batches - Low

`defer_batch` accepted an empty **`steps_batch`** and entries where **`step_count == 0`**. A zero-step entry does not mint value, but its presence indicates inconsistent backend input and can make the batch's accounting assumptions less trustworthy.

For a protocol-controlled oracle flow, a single invalid entry should be treated as a malformed batch rather than silently processed.

Fixed code: batch input validation

```
require!(!steps_batch.is_empty(), "Empty steps batch");

for (account_id, step_count) in steps_batch {
  require!(step_count != 0, "Step count must not be zero");

  if self.is_restricted(&account_id) {
    continue;
  }
}
```

Recommendation

Require a non-empty batch and reject zero-step entries before applying denylist skips or accounting changes.

Remediation - Fixed

The deferred minting path now rejects empty batches and zero-step entries before computing payouts or scheduling holding-contract callback work. Rejected zero-step batches do not advance **`steps_since_tge`**. Fixed in commit **3921733e72ba2e9b8a9e3cdd5a15c4241724e357**.

GUV-9 Deferred minting had no explicit max batch size - Low

`defer_batch` builds a cross-contract call payload and then reserves only a small fixed callback gas amount. Without an explicit **`MAX_BATCH_SIZE`**, an oversized batch can consume enough gas in the holding-contract call or callback path that the callback does not execute.

If the callback does not execute, tokens are not deposited to the claim contract and the oracle fee is not minted, while local pre-callback accounting can become inconsistent with the intended batch lifecycle.

Affected area: defer_batch size

```
fn defer_batch(&mut self, steps_batch: Vec<(AccountId, u32)>) -> PromiseOrValue<()> {
  // no explicit maximum length check before processing
}
```

Recommendation

Introduce a protocol-level **`MAX_BATCH_SIZE`** based on gas testing, reject oversized batches before state mutation, and keep a safety margin for callback execution. Document the maximum as part of the backend contract for oracle submissions.

Remediation - Fixed

The deferred minting entry point now rejects batches above the configured maximum before processing entries or scheduling cross-contract work. The configured maximum is **`135`** entries. Batches with **`136`** entries are rejected and leave **`steps_since_tge`** unchanged. Fixed in commit **b8f927f268af938563e429bfb17d02694875e5ea**.

Fixed code: maximum batch size

```
const MAX_BATCH_SIZE: usize = 135;

require!(
  steps_batch.len() <= MAX_BATCH_SIZE,
  "Batch size exceeds the maximum allowed"
);
```

GUV-10 Missing event emissions reduced auditability - Informational

Some state-changing operations did not emit the events that downstream monitors and indexers would expect. **`set_restricted`** changed the denylist without emitting a project-specific restriction event, and forced **`storage_unregister`** could burn a remaining token balance without emitting a NEP-141 **`ft_burn`** event from the wrapper method.

This did not directly break authorization or token accounting, but it reduced observability and made off-chain reconciliation harder.

Affected area: missing events

```
fn set_restricted(&mut self, account_id: &AccountId, is_restricted: bool) {
  if is_restricted {
    self.denylist.insert(account_id);
  } else {
    self.denylist.remove(account_id);
  }
}

fn storage_unregister(&mut self, force: Option<bool>) -> bool {
  self.token.internal_storage_unregister(force).is_some()
}
```

Recommendation

Emit a project-specific NEP-297 style event when restriction status changes, and emit NEP-141 burn events whenever forced unregister burns a token balance.

Remediation - Fixed

The contract now emits **`restriction_changed`** events for denylist updates and emits **`FtBurn`** when forced **`storage_unregister`** burns a balance. Fixed in commits **73edfc4a7729ca1d4b5757b7a3a9fd30d7365429** and **9255cb6f2b00d553ec50f3c5c8cfc6ec1f0a2613**.

Fixed code: restriction and burn events

```
Event::RestrictionChanged {
   account_id,
   is_restricted,
}
.emit();

self.token
   .internal_storage_unregister(force)
   .inspect(|(account_id, balance)| {
      FtBurn {
        owner_id: account_id,
         amount: (*balance).into(),
        memo: None,
      }
      .emit();
   })
   .is_some()
```

GUV-11 Overlapping deferred batches can reuse step ranges after rollback - Informational

`defer_batch` updates **`steps_since_tge`** immediately, before the holding-contract promise finishes. Each batch therefore reserves and prices a step range first, then later confirms minting or rolls the range back in the callback.

If a second **`defer_batch`** is submitted before the previous batch's callback has resolved, an earlier failed holding call can roll back **`steps_since_tge`** behind a later successful batch. This can make the same step range available again even though the later successful batch already used that range.

Example: overlapping deferred batch rollback

Start: steps_since_tge = 0

Batch A reserves range 0..100 and remains pending
Batch B reserves range 100..200 and later succeeds
Batch A's holding call fails
A's callback subtracts 100 steps, moving counter back to 100
Next batch starts from 100 and can reuse range 100..200

Since rewards decrease as **`steps_since_tge`** increases, reusing a range can calculate later rewards too generously compared to the intended monotonically increasing difficulty schedule.

Recommendation

If strict on-chain accounting is required, enforce one unresolved **`defer_batch`** at a time, make **`steps_since_tge`** monotonic by not rolling it back on failed holding calls, or track pending batch ranges independently so a failed callback cannot roll back behind later successful ranges.

Remediation - Acknowledged

The issue is acknowledged as an edge case. The current design prioritizes deferred-minting throughput and relies on operational handling of overlapping batches rather than enforcing single-flight batch processing on-chain.