Guvenkaya® The Bedrock of Security

The Sweat Foundation Ltd

Defer NEAR Rust Smart Contract Security Assessment

Lead Security Engineer: Timur Guvenkaya
Date of Engagement: 22nd January 2024 - 25th January 2024
Visit: www.guvenkaya.co

Contents

About Us

Guvenkaya is a security research firm specializing in Rust security, Web3 security of Rust-based protocols, and Web2 security. With our expertise, we provide both security auditing services and custom security solutions

About The Sweat Foundation

The Sweat Foundation is an organization behind Sweat Economy, an innovative project at the intersection of fitness and crypto. It motivates users to stay active by converting their steps into SWEAT Token. This approach promotes health and fitness and works as an entry point to crypto for many users.

Audit Results

Guvenkaya conducted a security assessment of the Sweat Economy Defer feature from 22nd January 2024 to 25th January 2024. During this engagement, a total of 4 findings were reported. 3 of the findings were low and 1 was informational severity. All the issues were fixed by the Sweat Foundation team.

Project Scope

Sweat FT

File name	Link
Defer	https://github.com/sweatco/sweat-near/blob/8fe03ae5da77574ee4b46f3e8936c4ea44bd5721/sweat/src/defer.rs

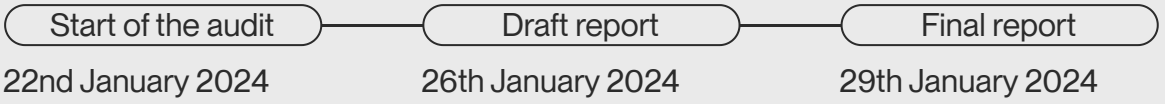
Sweat Claim

File name	Link
Api	https://github.com/sweatco/sweat-claim/blob/bf93665d86bb36dfab898e3788ebd5a1a09e52e6/contract/src/record/api.rs
Mod	https://github.com/sweatco/sweat-claim/blob/bf93665d86bb36dfab898e3788ebd5a1a09e52e6/contract/src/record/mod.rs

Out of Scope

The audit will include, but is not limited to, reviewing the code for security vulnerabilities, coding practices, and architecture. The audit does not include a review of the dependencies, **math.rs**, and token calculation functionality which uses math.rs.

Timeline



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

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 difcult 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: The vulnerability is easily discoverable and not overly complex to exploit.
Moderate: The vulnerability presents some challenges either in discovery or in the complexity of the attack.
Negligible: The vulnerability is either very difcult to discover or complex to exploit, or both.

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 moderate impact with a likely likelihood.
Medium: Used for vulnerabilities with severe impact but 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	Moderate	Negligible
Likely	**CRITICAL**	**HIGH**	**MEDIUM**
Possible	**HIGH**	**MEDIUM**	Low
Rare	**MEDIUM**	Low	Informational

Findings Summary

01. Remediation Complexity: This measures how difcult it is to fx the vulnerability once it has been identifed.

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 measures how difcult it is to fx the vulnerability once it has been identifed.

Not Fixed: Indicates that the vulnerability has been identifed 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 (like patching, confguration changes, or architectural modifcations) 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 fxed. It indicates that the risk posed by the vulnerability is known and has been consciously accepted.

Finding	Impact	Likelihood	Severity	Remediation Complexity	Remediation Status
GUV-1 - Race Condition Lock Is Not Asserted	Negligible	Possible	Low	Simple	Fixed
GUV-2 - Defer Batch Might Fail Under Production Batch Size	Negligible	Possible	Low	Simple	Fixed
GUV-3 Suboptimal Rounding Direction For An Oracle Fee	Negligible	Possible	Low	Simple	Fixed
GUV-4 Usage Of Custom Check Instead Of #[private] In The Callback	Negligible	Rare	Informational	Simple	Fixed

Findings Details

GUV-1 Race Condition Lock Is Not Asserted - Low

We've observed that the claim function uses the 'is_locked' field in the AccountRecord as a race condition lock to prevent issues such as double spending. However, despite the account being locked prior to making a cross-contract call, the lock is not asserted at the beginning of the function, rendering it ineffective.

sweat-claim:claim:contract/src/claim/api.rs

```
        fn claim(&mut self) -> PromiseOrValue<ClaimResultView> {
            let account_id = env::predecessor_account_id();

            require!(
                self.is_claim_available(account_id.clone()) == ClaimAvailabilityView::Available,
                "Claim is not available at the moment"
            );

            let account_data = self.accounts.get_mut(&account_id).expect("Account data is
not found");
            account_data.is_locked = true;
```

Fortunately, even without enabled lock, no race condition occurred because accruals were cleared before a cross-contract call. Therefore, even if the function is called again, the user doesn't have any accrued balance to claim. This was tested using both a batch call through near_workspaces and through a smart contract.

poc:batch

```
#[tokio::test]
async fn race_condition_exploit() -> anyhow::Result<()> {
    let mut context = prepare_contract().await?;

    let alice = context.alice().await?;
    let manager = context.manager().await?;

    let alice_steps = 10_000;
    let alice_initial_balance =
context.ft_contract().ft_balance_of(alice.to_near()).call().await?;

    let target_token_amount = context.ft_contract().formula(U64(0),
alice_steps).call().await?.0;
    let target_fee = target_token_amount * 5 / 100;
    let target_effective_token_amount = target_token_amount - target_fee;

    context
        .ft_contract()
        .defer_batch(vec![(alice.to_near(), alice_steps)], context.sweat_claim().account())
        .with_user(&manager)
        .call()
        .await?;

    let claim_contract_balance = context
        .ft_contract()
        .ft_balance_of(context.sweat_claim().account())
        .call()
        .await?;
```

poc:batch

```
    assert_eq!(claim_contract_balance.0, target_effective_token_amount);

    let alice_deferred_balance = context
        .sweat_claim()
        .get_claimable_balance_for_account(alice.to_near())
        .call()
        .await?;
    assert_eq!(alice_deferred_balance.0, target_effective_token_amount);

     alice.batch(&context.sweat_claim().account().to_string().parse()?)
         .call(Function::new("claim").args_json(json!({})))
         .call(Function::new("claim").args_json(json!({})))
         .transact().await?.into_result()?;

     let alice_balance = context.ft_contract().ft_balance_of(alice.to_near()).call().await?;

     let alice_balance_change = alice_balance.0 - alice_initial_balance.0;

     assert_eq!(alice_balance_change, target_effective_token_amount);

     Ok(())
}
```

poc:exploit_smart_contract

```
#[tokio::test]
async fn race_condition_exploit() -> anyhow::Result<()> {
    let mut context = prepare_contract().await?;

    let exploit_contract = context.worker.clone();

    let exploit_contract = exploit_contract.dev_deploy(EXPLOIT_CONTRACT).await?;

    context
        .ft_contract()
        .storage_deposit(Some(exploit_contract.id().clone().parse()?), None)
        .call()
        .await?;
    context
        .ft_contract()
        .tge_mint(&exploit_contract.id().clone().parse()?, U128(100_000_000))
        .call()
        .await?;

    let alice = exploit_contract.id().to_string().parse::<AccountId>()?;
    let manager = context.manager().await?;

    let alice_steps = 10_000;
    let alice_initial_balance =
context.ft_contract().ft_balance_of(alice.clone()).call().await?;

    let target_token_amount = context.ft_contract().formula(U64(0),
alice_steps).call().await?.0;
    let target_fee = target_token_amount * 5 / 100;
    let target_effective_token_amount = target_token_amount - target_fee;
```

poc:exploit_smart_contract

```
        context
            .ft_contract()
            .defer_batch(vec![(alice.clone(), alice_steps)], context.sweat_claim().account())
            .with_user(&manager)
            .call()
            .await?;

        let claim_contract_balance = context
            .ft_contract()
            .ft_balance_of(context.sweat_claim().account())
            .call()
            .await?;

        assert_eq!(claim_contract_balance.0, target_effective_token_amount);

        let alice_deferred_balance = context
            .sweat_claim()
            .get_claimable_balance_for_account(alice.clone())
            .call()
            .await?;

        assert_eq!(alice_deferred_balance.0, target_effective_token_amount);

        exploit_contract.call("call_claim").args_json(json!({"addr":
    context.sweat_claim().account()})).max_gas()
            .transact().await?.into_result()?;

        let alice_balance = context.ft_contract().ft_balance_of(alice).call().await?;
        let alice_balance_change = alice_balance.0 - alice_initial_balance.0;
        assert_eq!(alice_balance_change, target_effective_token_amount);

        Ok(())
    }
```

source:exploit_smart_contract

```
#[near_bindgen]
impl Contract {
    pub fn call_claim(addr: AccountId) -> Promise {
        Promise::new(addr.clone())
            .function_call(
                "claim".into(),
                json!({}).to_string().as_bytes().to_vec(),
                0,
                (ONE_TERRA * 100).into(),
            )
            .and(Promise::new(addr).function_call(
                "claim".into(),
                json!({}).to_string().as_bytes().to_vec(),
                0,
                (ONE_TERRA * 100).into(),
            ))
            .then(
                Self::ext(env::current_account_id())
                    .with_static_gas((ONE_TERRA * 30).into())
                    .resolve_callback(),
            )}
        pub fn resolve_callback(#[callback_result] result: Result<ClaimResultView,
PromiseError>) {
            match result {
                Ok(result) => {
                    log!("Success: {:#?}", result);
                }
                Err(e) => {
                    log!("Error: {:?}", e);
                }
            }
        }
    }
```

PROPOSED SOLUTION

We propose adding an assertion in the beginning of the claim function

REMEDIATION - FIXED

The Sweat Foundation team has fixed the issue by adding an assertion in the beginning of the claim function in this commit: **4077a6d51baad63a1bbb8bbfc7b34d6e2f70c9c3**

GUV-2 Defer Batch Might Fail Under Production Batch Size - Low

We've noticed that under certain conditions, the defer batch may fail with the production batch size. The backend utilizes a batch size of 135 when invoking the defer function in the FT smart contract. After FT makes a cross-contract call to the claim smart contract to record steps, it generates a log by pushing the (account_id, amount) tuples to the amounts field of the RecordData structure, for each batch entry. This pattern carries a risk of exceeding the log size limit, which could cause a transaction failure. Additionally, no tests use the actual production batch size. Therefore, while increasing the batch size might pass the tests, transactions could still fail in practice due to exceeding prepaid gas or the log size limit.

sweat_claim:record_batch_for_hold:contract/src/record/api.rs

```
let mut event_data = RecordData {
    timestamp: now_seconds,
  amounts: vec![],
};
for (account_id, amount) in amounts {
  event_data.amounts.push((account_id.clone(), amount));

    let amount = amount.0;
    let index = balances.len();

    total_balance += amount;
    balances.push(amount);
    ...

  emit(EventKind::Record(event_data));
```

We conducted tests with a constant number of 10,000 steps, which emulates a possible number of steps for each entry. We used accounts of varying lengths to determine the minimum account length required for a batch of 135 entries to fail. After testing, we found that the minimum account length is 57 characters, plus 5 characters from the .near suffix. This means if the backend sends a batch of 135 entries, each with an account length of 62 characters, the transaction will fail due to the log size limit.

poc:defer_max

```
#[tokio::test]
async fn defer_max() -> anyhow::Result<()> {
    let mut context = prepare_contract().await?;

    let manager = context.manager().await?;

    let long_account = format!("{}.near", "a".repeat(57))
        .parse::<AccountId>()?;

    let vec = vec![(long_account, 10000); 135];

    context
        .ft_contract()
        .defer_batch(vec, context.sweat_claim().account())
        .with_user(&manager)
        .call()
        .await?;

    Ok(())
}
```

PROPOSED SOLUTION

The likelyhood of this issue occurring in production is low, but we still propose to add a test with the production batch size to ensure that the issue does not occur in the future.

REMEDIATION - FIXED

The Sweat Foundation team has fixed the issue by adding a test with the production batch size in this commit: **09aa633d7dd408fa6ee8a6fee5604d7be8d85c1f**

GUV-3 Suboptimal Rounding Direction For An Oracle Fee - Low

During token calculations, it was observed that the oracle fee is rounded down. When the sweat_to_mint value is less than 20, the oracle fee is 0. In all other cases, the oracle fee is rounded down, which results in a cumulative loss for the protocol (even if it is insignificant). As SWEAT FT has 18 decimals, having a sweat_to_mint value of less than 20 is rare. However, it is recommended to always round against the user to prevent cumulative loss and any potential system exploitation.

sweat-near:calculate_tokens_amount:contract/src/record/api.rs

```
pub(crate) fn calculate_tokens_amount(&self, steps: u32) -> (u128, u128) {
    let sweat_to_mint: u128 = self.formula(self.steps_since_tge, steps).0;
  let trx_oracle_fee: u128 = sweat_to_mint * 5 / 100;
    let minted_to_user: u128 = sweat_to_mint - trx_oracle_fee;

    (minted_to_user, trx_oracle_fee)
}
```

PROPOSED SOLUTION

We propose to use div_ceil to round up the oracle fee.

REMEDIATION - FIXED

The Sweat Foundation team has fixed the issue by using div_ceil to round up the oracle fee in this commit: **e208f3a06050637c6477c73e06cd69a241fe5d7d**

GUV-4 Usage Of Custom Check Instead Of #[private] In The Callback - Informational

It was observed that the on_record callback uses a custom check to determine if the signer is an oracle to restrict access. However, this callback should not be directly accessed by anyone. Additionally, the signer account isn't utilized anywhere beyond the initial check. To avoid this and restrict the callback to the contract only, the callback should be marked as #[private]. This approach is more conventional and clarifies the intention.

sweat-near:on_record:contract/src/record/api.rs

```
        fn on_record(&mut self, receiver_id: AccountId, amount: U128, fee_account_id:
AccountId, fee: U128) {
          if !self.oracles.contains(&env::signer_account_id()) {
              panic_str("The operation can be only initiated by an oracle");
          }

          if !is_promise_success() {
              panic_str("Failed to record data in holding account");
          }

          let mut events: Vec<FtMint> = Vec::with_capacity(2);
          ...
        }
```

PROPOSED SOLUTION

We propose to use **#[private]** macro to restrict accesss to the callback.

REMEDIATION - FIXED

The Sweat Foundation team has fixed the issue by using #[private] macro in this commit: **7bbae6072f8b79fe92c9fff1770233bdb30a06e2**