Guvenkaya® The Bedrock of Security

The Sweat Foundation Ltd

Sweat Booster & Step Jars NEAR Rust Smart Contract Security Assessment

Lead Security Engineer: Timur Guvenkaya

Date of Engagement: 19th August 2024 - 28th August 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 Booster smart contract and the step jars feature inside the Sweat Jar smart contract. During this engagement, 6 findings were reported. 2 were Critical, 3 High, and 1 Medium severity. The Sweat Foundation team has fixed all issues.

Project Scope

Sweat Booster Contract

Commit Hash: **724df967363411092f12ed5467c20edc3320e8ef**

File name	Link
redeem/api.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/redeem/api.rs
common/nft_interface.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/common/nft_interface.rs
mint/model.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/mint/model.rs
burn/api.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/burn/api.rs
init/api.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/init/api.rs
mint/api.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/mint/api.rs

File name	Link
common/asserts.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/common/asserts.rs
common/ft_interface.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/common/ft_interface.rs
auth/api.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/auth/api.rs
lib.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/lib.rs
config/api.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/config/api.rs
init/model.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/init/model.rs
common/mod.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/common/mod.rs
mint/mod.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/mint/mod.rs
init/mod.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/init/mod.rs
redeem/mod.rs	https://github.com/sweatco/sweat-booster/blob/724df967363411092f12ed5467c20edc3320e8ef/contract/src/redeem/mod.rs

Sweat Jars Contract

Commit Hash: **04370ca62fe7ed61b264870851909e365f5bedcc**

File name	Link
score/account_score.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/contract/src/score/account_score.rs
score/charts.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/contract/src/score/charts.rs
score/api.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/contract/src/score/api.rs
score/mod.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/contract/src/score/mod.rs
migration/step_jars.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/contract/src/migration/step_jars.rs
claim/api.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/contract/src/claim/api.rs
withdraw/api.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/contract/src/withdraw/api.rs
udecimal.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/model/src/udecimal.rs

File name	Link
timezone/timestamps.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/model/src/timezone/timestamps.rs
timezone/timezone.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/model/src/timezone/timezone.rs
score.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/model/src/score.rs
timezone/mod.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/model/src/timezone/mod.rs
jar/model/common.rs	https://github.com/sweatco/sweat-jar/tree/04370ca62fe7ed61b264870851909e365f5bedcc/contract/src/jar/model/common.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. Only the changes related to the step jars are in scope for the Sweat Jar contract.

Timeline

Start of the audit	Draft report	Final report
19th August 2024	28th August 2024	19th December 2024

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: Missing Access Control On NFT Transfer Resolver	Severe	Likely	CRITICAL	Simple	Fixed
GUV-2: Missing Access Control On Account Score Recording	Severe	Likely	CRITICAL	Simple	Fixed
GUV-3: Per User DoS Of The Claim & Withdraw Functionality	Severe	Possible	HIGH	Complex	Fixed
GUV-4: DoS Of Record Score Functionality	Severe	Possible	HIGH	Complex	Fixed
GUV-5: Inconsistent State After NFT Burn	Severe	Possible	HIGH	Moderate	Fixed
GUV-6: Possible Scam Scenario Via Race Condition	Moderate	Possible	MEDIUM	Simple	Fixed
GUV-7: Score Is Not Reverted After Failed Transfer	Moderate	Possible	MEDIUM	Simple	Fixed

Findings Details

GUV-1 Missing Access Control On NFT Transfer Resolver - Critical

We observed that **nft_resolve_transfer** function lacks any access control. By using this function, a malicious actor can steal NFTs from any account.

common:nft_resolve_transfer:common/nft_interface.rs

```
        #[near]
        impl NonFungibleTokenResolver for Contract {
            fn nft_resolve_transfer(
                &mut self,
                previous_owner_id: AccountId,
                receiver_id: AccountId,
                token_id: TokenId,
                approved_account_ids: Option<HashMap<AccountId, u64>>,
            ) -> bool {
                self.tokens
                    .nft_resolve_transfer(previous_owner_id, receiver_id, token_id,
    approved_account_ids)}}
```

PROPOSED SOLUTION

Consider adding the **#[private]** macro on top of the function to make it only callable by the contract itself.

REMEDIATION - FIXED

The Sweat Foundation team has fixed the issue by adding the **#[private]** macro on top of the function in this commit: **fe94acf23ec7af8bd5d0ae26ec7db75af36f47dc**

GUV-2 Missing Access Control On Account Score Recording - Critical

We observed that **record_score** function lacks any access control. By using this function, a malicious actor can put any score to its account and then claim the reward.

score:record_score:contract/src/score/api.rs

```
#[near_bindgen]
impl ScoreApi for Contract {
  fn record_score(&mut self, batch: Vec<(AccountId, Vec<(Score, UTC)>)>) {
    let mut event = vec![];

    let now = block_timestamp_ms();

    for (account, new_score) in batch {
      self.migrate_account_jars_if_needed(&account);
      ...
      }
 }
```

PROPOSED SOLUTION

Consider asserting that only manager can call this function.

REMEDIATION - FIXED

The Sweat Foundation team has fixed the issue by asserting whether the caller is the manager in this commit: **f09a5d482e02726ffaa38f4f5b1bbfaf8c3261ad**

GUV-3 Per User DoS Of The Claim & Withdraw Functionality - High

It was observed that it is possible for a malicious actor to trigger DoS of claim and withdraw functionality by locking jars of other users. During the withdrawal and claiming process, account jars are temporarily locked to prevent race condition issues. When callback is reached, the lock is released. However, if there was a panic in the callback function, the lock would not be released, causing jars to be locked forever. In return, it won't allow users to claim or withdraw from them. In the current implementation, it is possible to get the panic in the callback by reaching the log limit since at the end of the function the log is emitted containing all of the jars that were processed. In addition, it is possible to create a jar for someone else and there are no limit in the number of jars that can be created per account. These scenarios open up the possibility of this attack on each user:

1. 1. Pick a victim account.
2. 2. Create a ticket with the receiver and get the signature.
3. 3.Do ft_transfer_call with a staking message to create a jar for receiver.
4. 4. Repeat 2 and 3 until you create around 470 jars (enough to cause log limit)
5. 5. Wait until the user tries to claim or withdraw all
6. 6. Now all of the victim's jars are locked

POC

poc:per_user_dos

```
#[tokio::test]
    async fn per_user_dos() -> anyhow::Result<()> {
        let mut context = prepare_contract(
            None,
            [
                RegisterProductCommand::Locked12Months12Percents,
                RegisterProductCommand::Locked6Months6Percents,
                RegisterProductCommand::Locked6Months6PercentsWithWithdrawFee,
                RegisterProductCommand::Locked5Minutes60000Percents,
            ],
        )
        .await?;
```

poc:per_user_dos (continued)

```
    let alice = context.alice().await?;
    let manager = context.manager().await?;
    let products = context.sweat_jar().get_products().await?;
    assert_eq!(4, products.len());

    context
        .sweat_jar()
        .create_jar(
            &alice,
            RegisterProductCommand::Locked12Months12Percents.id(),
            1_000_000,
            &context.ft_contract(),
        )
        .await?;
     let alice_principal = context.sweat_jar().get_total_principal(alice.to_near()).await?;
     let alice_interest = context.sweat_jar().get_total_interest(alice.to_near()).await?;
     assert_eq!(1_000_000, alice_principal.total.0);
     assert_eq!(0, alice_interest.amount.total.0);
     context.fast_forward_hours(1).await?;
     context
        .sweat_jar()
        .bulk_create_jars(
            alice.to_near(),
            RegisterProductCommand::Locked5Minutes60000Percents.id(),
            10000,
            500,
        )
        .with_user(&manager)
        .await?;
```

poc:per_user_dos (continued)

```
     context.fast_forward_minutes(5).await?;
         context
           .sweat_jar()
           .claim_total(true.into())
           .with_user(&alice)
           .await
           .expect_err("The length of a log message 16440 exceeds the limit 16384");
         // CLAIMING AGAIN BUT ALL JARS ARE LOCKED
         let claimed_details =
   context.sweat_jar().claim_total(Some(true)).with_user(&alice).await?;
         let ClaimedAmountView::Detailed(claimed_details) = claimed_details else {
           panic!()
         };
         let claimed_amount = claimed_details.total.0;
         assert_eq!(claimed_amount, 0);
         // Cannot withdraw
         context
           .sweat_jar()
           .withdraw(U32(1), None)
           .with_user(&alice)
           .expect_error("Another operation on this Jar is in progress")
           .await?;
         Ok(())
       }
```

PROPOSED SOLUTION

We propose switching to having a single jar per product to avoid any types of DoS attacks due to limitless jars.

REMEDIATION - FIXED

The Sweat Foundation refactored the jars codebase to have a single jar per product in this PR: https://github.com/sweatco/sweat-jar/pull/114

GUV-4 DoS Of Record Score Functionality - High

We observed the **record_score** iterates over user jars to then perform interest calculation. However, since there is no limit on the number of jars per account, it is possible for an attacker to create a large number of jars for a single account and then when oracle calls the record_score function it will lead to a panic due to the gas limit. This will cause DoS of record_score every time there is a malicious account in a batch.

score:record_score:contract/src/score/api.rs

```
        for jar in &mut account_jars.jars {
            let product = self
                .products
                .get(&jar.product_id)
                .unwrap_or_else(|| env::panic_str(&format!("Product '{}' doesn't exist",
    jar.product_id)));

            if !product.is_score_product() {
                continue;
            }

            let (interest, remainder) = jar.get_interest(&score, &product, now);
            ...
        }
```

PROPOSED SOLUTION

We propose switching to having a single jar per product to avoid any types of DoS attacks due to limitless jars.

REMEDIATION - FIXED

The Sweat Foundation refactored the jars codebase to have a single jar per product in this PR: https://github.com/sweatco/sweat-jar/pull/114

GUV-5 Inconsistent State After NFT Burn - High

We observed that the booster smart contract will have an inconsistent state if a user minted more than 1 token and then redeemed/burned. That's because even if a user has more than one token, the whole entry of **tokens_per_owner is** removed. If minted again, the nested structure will show that it has a length of 1, for instance, even though when we manually call **.contains()** it will include 2 tokens.

burn:burn:contract/src/burn/api.rs

```
fn burn(&mut self, token_id: TokenId) -> TokenMetadata {
  self.owner_by_id.remove(&token_id).expect("Owner not found");

    if let Some(approvals_by_id) = &mut self.approvals_by_id {
        approvals_by_id.remove(&token_id);
    }

    self.token_metadata_by_id
        .as_mut()
        .and_then(|by_id| by_id.remove(&token_id))
        .expect("Token metadata not found")
}
```

POC

poc:burn_inconsistent_state

```
#[test]
  fn burn_token_inconsistent_state() {
    let oracle = oracle();
    let mut context = Context::new(&oracle);
    let media =
"bafkiufjkssmvjby7srbr44ivexsexqvdvjby7hhoobl3q3twim3bgxffrm".to_string();
    let media_hash =
Base64VecU8::from(b"w5SZrIcpoakfcRUI5EvCa4hy2ZzscFe4bnZDNhNcpYs=".to_vec());
    let denomination = U128(2_000_000);
    context.switch_account(oracle);
    // MINTING 1st TOKEN
    context.with_deposit_yocto(DEPOSIT_FOR_MINTING, |context| {
      context
        .contract()
        .mint_balance_booster(alice(), denomination.clone(), media.clone(),
media_hash.clone());
    });
    // MINTING 2nd TOKEN
    context.with_deposit_yocto(DEPOSIT_FOR_MINTING, |context| {
      context
        .contract()
        .mint_balance_booster(alice(), denomination.clone(), media.clone(),
media_hash.clone());
    });
    let alice_tokens = context.contract().nft_tokens_for_owner(alice(), None, None);
    assert_eq!(2, alice_tokens.len());
    let token_to_burn = alice_tokens.first().unwrap();
    context.contract().burn(alice(), token_to_burn.token_id.clone());
    let burnt_token = context.contract().nft_token(token_to_burn.token_id.clone());
    assert!(burnt_token.is_none());
```

poc:burn_inconsistent_state (continued)

```
        // NONE EVEN THOUGH WE HAVE 1 UNBURNED TOKEN LEFT
        assert!(context
            .contract()
            .tokens
            .tokens_per_owner
            .as_ref()
            .unwrap()
            .get(&alice())
            .is_none());
        // MINTING ANOTHER ONE
        context.with_deposit_yocto(DEPOSIT_FOR_MINTING, |context| {
            context
                .contract()
                .mint_balance_booster(alice(), denomination, media, media_hash);
        });
        let alice_data = context
        .contract()
        .tokens
        .tokens_per_owner
        .as_ref()
        .unwrap()
        .get(&alice())
        .unwrap();
    // LEN IS 1 EVEN THOUGH WE HAVE 2 TOKENS
    assert_eq!(alice_data.len(), 1);
    // CONTAINS 1 AND 2 EVEN THOUGH THE LEN IS 1
    assert!(alice_data.contains(&"1".to_string()));
    assert!(alice_data.contains(&"2".to_string()));
}
```

PROPOSED SOLUTION

We propose removing the outer entry only if the nested entry is empty. Otherwise, the nested entry has to be updated and inserted back.

REMEDIATION - FIXED

The Sweat Foundation team has fixed the issue by removing the outer entry if the internal one is empty: **ebe5d8012d56907b8db261c301d9699a7916aec2**

GUV-6 Possible Scam Scenario Via Race Condition - Medium

We observed that it is possible to perform NFT transfer operations while the redeem process is in progress. After redeem process is done, in the callback the NFT is burned. This situation opens an opportunity for a malicious actor to scam another user/system.

Example: Alice wants to redeem SWEAT voucher in the third party and receive any other token in return. That third party maybe reselling voucher or doing something else with that NFT (NFT is needed for them)

1. 1. Alice initiates a call to redeem the NFT voucher.
2. 2. At the same time, Alice calls **nft_transfer** to transfer NFT to the third party.
3. 3.Alice receives tokens or something else in return from the third party
4. 4. Callback is reached after 2 blocks and that NFT is burned from the third party.

Result: Alice received something from the third party but another side received nothing since their NFT got burned.

PROPOSED SOLUTION

We propose checking whether the redeem is in progress inside of the **nft_transfer** and the **nft_transfer_call**.

REMEDIATION - FIXED

The Sweat Foundation team has fixed the issue by including a check which prevents calling nft transfer function during the redeem process:
cc4da86d967771c776a70c223d9fd26495484245

GUV-7 Score Is Not Reverted After Failed Transfer - Medium

We observed that in the **claim_jars_internal**, the score is set to zero after fetching the score to get the interest. After the transfer is done, if the transfer fails, the score is not reverted to the previous value inside of the **after_claim_internal** callback. If a user tries to repeat the same operation, the user score will remain zero, which leads to the incorrect interest calculation for that user.

claim:claim_jars_internal:contract/src/claim/api.rs

```
fn claim_jars_internal(
        &mut self,
        account_id: AccountId,
        jar_ids: Vec<JarIdView>,
        detailed: Option<bool>,
    ) -> PromiseOrValue<ClaimedAmountView> {
        let now = env::block_timestamp_ms();
        let mut accumulator = ClaimedAmountView::new(detailed);

        let unlocked_jars: Vec<Jar> = self
            .account_jars(&account_id)
            .iter()
            .filter(|jar| !jar.is_pending_withdraw && jar_ids.contains(&U32(jar.id)))
            .cloned()
            .collect();

        let mut event_data: Vec<ClaimEventItem> = vec![];

        let account_score = self.account_score.get_mut(&account_id);

        let score =
account_score.map(AccountScore::claim_score).unwrap_or_default();

        for jar in &unlocked_jars {
            ...
```

score:claim_score:contract/src/score/account_score.rs

```
/// On claim we need to clear active scores so they aren't claimed twice or more.
    pub fn claim_score(&mut self) -> Vec<Score> {
        let today = self.timezone.today();

        let result = if today == self.update_day() {
            let score = self.scores[1];
            self.scores[1] = 0;
            vec![score]
        } else {
            let score = vec![self.scores[0], self.scores[1]];
            self.scores[0] = 0;
            self.scores[1] = 0;
            score
        };

        self.updated = block_timestamp_ms().into();

        result
    }
```

PROPOSED SOLUTION

We propose reverting the state of the score to the previous value in the **after_claim_internal**.

REMEDIATION - FIXED

The Sweat Foundation team has fixed the issue by reverting the score in the callback:
6967bccb40eaa8f5b36494f99c2408ad359f0b49