Guvenkaya® The Bedrock of Security

The Sweat Foundation Ltd

Jars Refactor Rust Smart Contract Security Assessment

Lead Security Engineer: Timur Guvenkaya

Date of Engagement: 9th December 2024 - 18th December 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 Jars smart contract refactor. During this engagement, 3 findings were reported. 1 was High ans 2 Medium severity . The Sweat Foundation team has fixed all the issues.

Project Scope

File name	Link
Lib	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/lib.rs
Internal	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/internal.rs
FT Receiver	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/ft_receiver.rs
FT Interface	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/ft_interface.rs
Event	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/event.rs
Assert	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/assert.rs
Claim Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/claim
Withdraw Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/withdraw
Restake Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/restake

File name	Link
Score Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/score
Product Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/product
Penalty Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/product
Migration Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/product
Jar Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/product
Fee Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/product
Common Functionality	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/contract/src/product
Model	https://github.com/sweatco/sweat-jar/tree/ba110ce01f9a104e7d533b24bafe57956976f2a9/model/src

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.

Timeline

Start of the audit	Draft report	Final report
9th December 2024	19th December 2024	10th January 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: Potential DoS of Batch Operations	Severe	Possible	HIGH	Moderate	Fixed
GUV-2: Missing Race Condition Lock	Moderate	Possible	MEDIUM	Simple	Fixed
GUV-3: Missing Score Product Key Enforcement During Creation	Moderate	Possible	MEDIUM	Simple	Fixed
GUV-4: Suboptimal Rounding Direction	Negligible	Possible	Low	Simple	Fixed
GUV-5: Outdated Gas Measurements	Negligible	Rare	Informational	Simple	Fixed
GUV-6: Redundant State Access	Negligible	Rare	Informational	Simple	Fixed

Findings Details

GUV-1 Potential DoS of Batch Operations - High

Before a user can create a deposit to a jar, the amount is validated against minimum and maximum caps.

jars:assert_cap:contract/src/product/model/v1.rs

```
pub(crate) fn assert_cap(&self, amount: TokenAmount) {
        if self.cap.min > amount || amount > self.cap.max {
          env::panic_str(&format!(
            "Total amount is out of product bounds: [{}..{}]",
            self.cap.min, self.cap.max
          ));
        }
      }
```

After discussion with the team, we discovered that the minimum cap in production is **1 SWEAT**. This minimum deposit amount is too low, allowing an attacker to create numerous deposits for any user. This can trigger a gas limit failure in methods like **withdraw_all**, **claim_total**, and **restake_all**, preventing users from performing these operations.

The attack requires approximately **$30 in SWEAT** and **4,500 deposits**. The vulnerability affects all batch operations because the system iterates over all products—even if a user has other products with higher minimum deposits, operations like restake_all, claim_total, and withdraw_all will fail due to the product with excessive deposits.

Recommendation

Consider raising the minimum deposit amount to make the attack economically unviable.

Remediation - Fixed

The Sweat Foundation team has fixed the issue by throttling requests on the backend side, setting keys for all products, add monitoring for jars' creation and stop issuing tickets for suspicious users

GUV-2 Missing Race Condition Lock - Medium

The **restake** method is missing an **is_pending_withdraw** lock on jars. This lock prevents race conditions before the callback arrives after a cross-contract call. When a jar has is_pending_withdraw set to true, the operation is either skipped or the smart contract returns an error.

jars:assert_not_locked:contract/src/assert.rs

```
pub(crate) fn assert_not_locked(jar: &Jar) {
        require!(!jar.is_pending_withdraw, "Another operation on this Jar is in
progress");
    }
```

This issue could lead to a scenario where a user's deposit is recorded with a timestamp from the past inside of the cache instead of the actual update time. As a result, interest may begin accumulating earlier than intended.

Possible Scenario:

- User deregisters their account on the token contract
- Calls claim_total, which sets the cache to current timestamp
- When the callback arrives, it fails due to the unregistered account and rolls back the cache to the previous timestamp while retaining the new deposit.

Recommendation

Consider implementing the **is_pending_withdraw** check in the restake method.

Remediation - Fixed

The Sweat Foundation team has fixed the issue by adding the race condition lock in this commit: 1b0c87287cd553c7c210acafbfc3c1d3bd63e131

GUV-3 Missing Score Product Key Enforcement During Creation - Medium

Score-based products are premium products that require a signature. However, during product creation through the **register_product** method, the public key requirement is not enforced. This allows users to register score-based products without a public key, enabling deposits without any signature verification.

jars:register_product:contract/src/product/api.rs

```
    fn register_product(&mut self, command: ProductDto) {
            self.assert_manager();
            assert_one_yocto();

            assert!(self.products.get(&command.id).is_none(), "Product already exists");

            let product: Product = command.into();

            product.assert_fee_amount();

            self.products.insert(&product.id, &product);

            emit(EventKind::RegisterProduct(product));
        }
```

Recommendation

Consider enforcing public key validation in **register_product** when registering score-based products.

Remediation - Fixed

The Sweat Foundation team has fixed the issue by asseting the public key existence on score based products in this commit:

ba110ce01f9a104e7d533b24bafe57956976f2a9

GUV-4 Suboptimal Rounding Direction - Low

In the **restake_all** method's withdrawal fee calculation, the withdrawal fee is rounded down. This rounding behavior results in a cumulative loss of withdrawal fees for the protocol, though the impact is minimal.

jars:restake_all:contract/src/restake/api/api.rs

```
 if withdrawal_amount > 0 {
                let withdrawal_fee = total_fee * withdrawal_amount /
total_mature_balance;
                request.withdrawal = WithdrawalDto {
                    amount: withdrawal_amount,
                    fee: withdrawal_fee,
                };

                self.transfer_remainder(request, event)
```

Recommendation

To prevent cumulative losses and potential system exploitation, fees should always be rounded against the user. This can be achieved by using **div_ceil** to round up the **withdrawal_fee**.

Remediation - Fixed

The Sweat Foundation team has fixed the issue by utilizing **div_ceil** in this commit:
9bd95fc264d2dd286082f7d704270accef54dcca

GUV-5 Outdated Gas Measurements - Informational

After a recent refactor, the gas measurements for the **restake_all** operation have become outdated. This could lead to incorrect assumptions about gas consumption.

jars:measure_restake_all:integration-tests/src/measure/restake_all.rs

```
async fn measure_restake_all() -> Result<()> {
        set_integration_logs_enabled(false);

        let product = RegisterProductCommand::Locked5Minutes60000Percents;
        let mut context = prepare_contract(None, [product]).await?;
        let alice = context.alice().await?;

        for _ in 0..200 {
            add_jar(&context, &alice, product, 10_000).await?;
        }

        context.fast_forward_minutes(6).await?;

        context.sweat_jar().claim_total(None).with_user(&alice).await?;

        let gas = context
            .sweat_jar()
            .restake_all(product.get().id, None)
            .with_user(&alice)
            .result()
            .await?
            .total_gas_burnt;
        dbg!(pretty_gas_string(gas));

       //  1 jar -  6 TGas 225 GGas total:  6225437862976
       // 100 jars - 50 TGas 709 GGas total: 50709431315947
       // 200 jars - 86 TGas 607 GGas total: 86607517267105...}
```

Recommendation

Consider re-running the measurements and updating comments

Remediation - Fixed

The Sweat Foundation team has fixed the issue by re-running the measurements.

GUV-6 Redundant State Access - Informational

The code contains a redundant call to **get_or_create_account_mut**.

jars:measure_restake_all:integration-tests/src/measure/restake_all.rs

```
let account = self.get_or_create_account_mut(&account_id);

            if signature.is_some() {
                account.nonce += 1;
            }

            if matches!(product.terms, Terms::ScoreBased(_)) {
                account.try_set_timezone(ticket.timezone);
            }

           let account = self.get_or_create_account_mut(&account_id);

            account.deposit(product_id, amount, None);

            emit(Deposit((product_id.clone(), amount.into())));
```

Recommendation

Remove the second **get_or_create_account_mut** call since the account reference is already available.

Remediation - Fixed

The Sweat Foundation team has fixed the issue by removing the redundant call in this commit: d409a301b5900353245d68f113aac27925b52a07