Guvenkaya® The Bedrock of Security

Sailor Lend

Sailor Lend NEAR Smart Contract Security Review

Lead Security Engineer: Michal Bajor
Date of Engagement: 10th December 2025 - 29th December 2025
Visit: www.guvenkaya.co

Contents

About Us

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

About Sailor Lend

Sailor Lend is a trustless, self-custodial DeFi lending protocol built on the NEAR blockchain. The platform enables users to obtain instant loans using BTC, ZEC, or SOL as collateral without requiring KYC verification. Users can deposit stablecoins for lending to earn interest, while borrowers can leverage their crypto assets to access liquidity. The protocol implements liquidation mechanisms, interest rate calculations, and Pyth price oracle integrations.

Audit Results

Guvenkaya conducted a comprehensive security assessment of the Sailor Lend NEAR smart contracts. During this engagement, 17 findings were reported: 5 Critical, 4 High, 5 Medium, and 3 Informational severity issues. The critical findings include race conditions in borrow and withdraw functionality enabling token theft, missing access controls on protocol configuration, unrestricted unlend allowing balance manipulation, and cross-contract call vulnerabilities that can drain the contract's native NEAR balance. All findings have been addressed by the Sailor Lend team through a full redesign and new financial model of the smart contract. Guvenkaya can confirm that the originally reported issues are fixed, but cannot attest whether the full redesign has not introduced new issues.

Project Scope

Sailor Lend Contract

Files	Link
Sailor Lend Lending Contract	https://github.com/Satoshi-Port/satoshi-port/tree/279250258752560a33e4aa7ac7a52d6f58fcb560/contract

Out of Scope

The audit included reviewing the code for security vulnerabilities, coding practices, and architecture. The audit does not include a review of the dependencies or external contracts such as Pyth Oracle and FT contracts.

Timeline

Start of the audit	Draft report	Final report
10th December 2025	29th December 2025	17th March 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

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: Exploitation could result in critical loss or compromise, such as full system control, substantial financial loss, or severe reputational damage.
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 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.

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: Protocol Config Lacks Access Control	Severe	Likely	CRITICAL	Simple	Fixed
GUV-2: Race Condition in Borrow Functionality	Severe	Likely	CRITICAL	Moderate	Fixed
GUV-3: Race Condition in Withdraw Function	Severe	Likely	CRITICAL	Moderate	Fixed
GUV-4: Token Theft Through Unrestricted Unlend	Severe	Likely	CRITICAL	Simple	Fixed
GUV-5: Cross-Contract Calls Drain Contract Balance	Severe	Likely	CRITICAL	Simple	Fixed
GUV-6: Leveraged Lending Is Possible	Severe	Possible	HIGH	Moderate	Fixed
GUV-7: Multiple Issues With Liquidation Mechanism	Severe	Possible	HIGH	Difficult	Fixed
GUV-8: Lack of Storage Staking Leads to DoS	Severe	Possible	HIGH	Difficult	Fixed
GUV-9: Overrepayment Is Not Properly Tracked	Moderate	Likely	HIGH	Simple	Fixed
GUV-10: Owner Can Arbitrarily Set Asset Price	Severe	Rare	MEDIUM	Simple	Fixed
GUV-11: Withdraw Callback Does Not See Errors	Moderate	Possible	MEDIUM	Moderate	Fixed
GUV-12: Invalid Approach to Fees and Interest	Moderate	Possible	MEDIUM	Moderate	Fixed
GUV-13: No Access Control in Notification Modification	Negligible	Likely	MEDIUM	Simple	Fixed

Finding	Impact	Likelihood	Severity	Remediation Complexity	Remediation Status
GUV-14: update_accounts_at_risk Might Fail to Execute	Moderate	Possible	**MEDIUM**	Moderate	Fixed
GUV-15: Dead or Useless Code	Negligible	Rare	Informational	Simple	Fixed
GUV-16: Invalid ft_transfer Parameters in Borrow	Negligible	Rare	Informational	Simple	Fixed
GUV-17: Logging Inconsistencies	Negligible	Rare	Informational	Simple	Fixed

Findings Details

GUV-1 Protocol Config Lacks Access Control - Critical

The **update_minimum_borrow_rate** function is responsible for setting the **minimum_borrow_rate** parameter, which is used to calculate interest rates within the protocol. We have observed that this function lacks any kind of access control validation and consequently is callable by anyone. As a result, any user can change the **minimum_borrow_rate** value for the whole protocol at any time.

core:update_minimum_borrow_rate

```
pub fn update_minimum_borrow_rate(&mut self, new_value: Decimal) {
  self.minimum_borrow_rate = new_value;
}
```

Recommendation

We recommend implementing an access control mechanism for **update_minimum_borrow_rate** using an RBAC system.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-2 Race Condition in Borrow Functionality - Critical

NEAR execution VM implements asynchronous cross-contract calls. This means that those calls, and any subsequent callback, do not happen in the same block, there is always a delay of at least one block.

Sailor Lend's borrow functionality implements a state change in the callback, seemingly to make sure that state changes only after successful transfers. However, this is only a partially correct approach. Without changing the state in the original execution, the contract is left susceptible to race conditions.

Based on the implementation, two security issues originate from it:

- User can borrow significantly more than his collateral allows him to.
- **self.total_lendable_supply -= amount** can cause **total_lendable_supply** to become negative, breaking the contract's accounting.

borrow.rs

```
fn internal_borrow(&mut self, amount: Decimal, token_id: TokenId, account_id: &AccountId)
-> Promise {
   // ... transfer args setup ...
   Promise::new(usdc_contract)
      .function_call("storage_deposit".to_owned(), storage_deposit_args, ...)
      .function_call("ft_transfer".to_owned(), transfer_args, ...)
      .then(
         Self::ext(env::current_account_id())
            .handle_borrow_callback(account_id, &token_id, amount),
      )
}

// State changes only happen in callback - vulnerable to race condition
pub fn handle_borrow_callback(&mut self, account_id: &AccountId, token_id:
&TokenId, amount: Decimal) {
   // ... balance updates happen here, after the transfer ...
}
```

Recommendation

We recommend changing the implementation so that:

1. User's borrowed balance is modified during the original **borrow** function execution. This way, each subsequent **borrow** execution, even if called in a batch transaction will be able to take into account that borrow.
2. The **handle_borrow_callback** function should revert the storage changes done in a previous step if the **ft_transfer** was not successful.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-3 Race Condition in Withdraw Function - Critical

NEAR execution VM implements asynchronous cross-contract calls. This means that those calls, and any subsequent callback, do not happen in the same block, there is always a delay of at least one block.

The Sailor Lend contract implements a withdrawal functionality which schedules the token transfer via **ft_transfer_call** flow and attaches a **handle_withdraw_callback** callback which performs actual internal accounting changes after a successful transfer.

Based on this implementation, user can actually withdraw more than the protocol-calculated max withdrawal amount by creating batch transactions with multiple withdraw calls.

core:withdraw

```
pub fn withdraw(&mut self, token_id: TokenId, amount: Decimal) {
   // ... validation checks pass for each call in batch ...
}

// Balance only updated in callback - vulnerable to race condition
pub fn handle_withdraw_callback_success(&mut self, account_id: &AccountId,
token_id: &TokenId, amount: Decimal) {
   let account = ensure_some(self.accounts.get_mut(account_id), "Account not
found");
   // ... withdraw from balance happens here ...
}
```

Recommendation

We recommend changing the implementation so that:

1. Internal balance accounting is modified to reflect the withdrawal during the initial **withdraw** function execution. That way, even when called in a batch transaction, each subsequent action will be going through the withdrawal checks that already take into account previous withdrawal attempts.
2. The callback should revert the balance changes on a failed token transfer.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-4 Token Theft Through Unrestricted Unlend - Critical

The Sailor Lend protocol allows depositing stablecoins that are then used for lending within the protocol. We have observed that it is possible to unlend more than user originally lent, provided that the **total_lendable_supply** is not smaller than an attempted unlend amount.

An attacker could lend some amount of stablecoin and wait for other users to do the same. Next, an attacker can call **unlend** function with **amount** parameter set to a higher value than original deposit. This will move attacker's **lent** balance into a negative value, but it will still modify the **unlent** balance to a specified amount. The **unlent** balance can then be used for withdrawal effectively stealing tokens from other users.

core:unlend

```
pub fn unlend(&mut self, token_id: String, amount: Decimal) -> Decimal {
  if self.total_lendable_supply < amount {
     env::panic_str("Critical error. The protocol doesn't have enough lendable supplies");
  }
  self.internal_unlend(amount, token_id.clone(), &account_id);
  amount
}

fn internal_unlend(&mut self, amount: Decimal, token_id: TokenId, account_id: &AccountId)
{
  let account = ensure_some(self.accounts.get_mut(account_id), "Account not found");
  let _new_lent = ensure_ok(
    account.state.lent_stablecoins.withdraw(token_id.clone(), amount),
    "Error during lending while deducting unlent amount",
  );
  let _new_unlent = ensure_ok(
    account.state.unlent_stablecoins.deposit(token_id.clone(), amount),
     "Error during lending while depositing lent amount",
  );
}
```

Recommendation

We recommend introducing checks that will prevent users from unlending more than their actual **lent** balance.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-5 Cross-Contract Calls Drain Contract Balance - Critical

It was observed that in many places throughout the codebase, the Sailor Lend contract is scheduling cross-contract calls to external contracts - FTs or Price Oracle. It was observed that it attaches a deposit with those calls to either cover possible storage deposit in FT contracts or pay for the price update in Pyth contract. However, the functions that are scheduling those calls are not expecting those tokens to be attached to them. As a consequence, the native NEAR tokens are deducted from the Sailor Lend contract's own free balance.

The consequences differ depending on the contract:

1. For Pyth Oracle, it means that anyone can use the **update_price_oracle** function to update any price data in the Pyth Oracle for free (Sailor Lend contract pays for the update)
2. For Fungible Tokens, it means that users can withdraw their storage stake from the FT contract, use the Sailor Lend contract to pay for their storage within the Contract. They can do it many times, effectively stealing native NEAR tokens from Sailor Lend contract.

Recommendation

We recommend introducing the following changes to the contract:

1. Make the **update_price_oracle** function payable and assert that funds required to cover the price update are provided by the caller.
2. Remove the calls to the **storage_deposit** functions within flows that would be paying for user's storage fees. Those should be covered by users themselves by design. Alternatively, Sailor Lend contract could assure that those funds are provided with the call, however this would increase the complexity of the contract.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-6 Leveraged Lending Is Possible - High

The Sailor Lend protocol allows users to deposit and lend stablecoins (to earn interest on their assets) and also to deposit crypto assets as collateral to use for borrowing stablecoins. It was observed that the protocol allows users to use borrowed assets for lending and create a leveraged position based on this.

An exemplary scenario:

1. User A deposits 1000 USDC and lends it all in the protocol.
2. User B deposits some substantial collateral, for example worth 100,000 USDC. As per the protocol's default configuration, user B can borrow at most 50,000 USDC.
3. User B borrows all of the 1000 USDC that's available in the protocol, then deposits that 1000 USDC himself and lends it again to the protocol.
4. User B can keep doing step 3, building a substantial lending position (up to 50,000 USDC) while only the initial 1000 USDC was actually in circulation. Any interest calculated would be using the 50,000 USDC as capital.

As long as interest earned through lending is higher than the fees associated with borrowing, User B can do this to exploit the protocol.

Recommendation

We recommend implementing interest calculation to be done on net lent balance. The actual capital used to calculate interest would be equal to **lent_amount - borrowed_amount**. The interest should be paid on net-positive lending positions. At the same time, fees calculated for the borrowing position should be charged on the full borrowed amount.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-7 Multiple Issues With Liquidation Mechanism - High

The liquidation mechanism is meant to keep the protocol's overall health good. However, multiple issues were identified:

1. No-one actually pays for the debt, i.e. protocol does not get the borrowed assets back. Because of this, liquidator also does not get any collateral in return. This makes it economically undesirable for anyone, apart from Protocol's owners, to perform a liquidation.
2. The liquidation itself is done by simply arbitrarily withdrawing all of the collateral supplied by liquidated user from his balance. However, this collateral is not transferred anywhere else (i.e. it does not go to a liquidator or to protocol's treasury), it is just no longer tracked by the protocol.
3. The prices used for liquidation are taken from the storage directly and no staleness check is executed. Consequently, if prices were not updated in a while, liquidation can be done using an outdated price.
4. The **execute_liquidation** function's docstrings explicitly mention that this function "liquidates enough collateral to bring the account back to a safe LTV". However, the actual implementation explicitly says that all of the deposited collateral is always liquidated.

Recommendation

We recommend implementing a staleness check for the prices used for liquidation and a review of the liquidation design.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-8 Lack of Storage Staking Leads to DoS - High

NEAR smart contracts are expected to pay for the storage they use. Those storage fees are taken automatically from the contract's free balance based on how many bytes had to be allocated after a storage usage increase within a given function execution. It is considered a best practice in NEAR ecosystem to implement storage staking, i.e. the storage allocated for data that relates to a given user has to be covered by that user.

We observed that the Sailor Lend contract does not implement any form of storage staking, neither for user's data, nor for the storage changes associated with the protocol's operation itself.

As a consequence, there will be a point in which contract's free balance will not be enough to cover the storage fees and every subsequent call that would try to allocate new storage will fail. This is a Denial-of-Service condition, although it is recoverable through sending more native NEAR tokens to the contract. This solution, however, is not sustainable and scalable.

Recommendation

We recommend implementing a robust storage staking mechanism that will enforce users to sign up and pay for their storage before allocating it for their data. Any function that increases the storage size used by contract should make sure the storage fees are covered.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-9 Overrepayment Is Not Properly Tracked - High

The Sailor Lend contract allows borrowing stablecoins and then naturally expects them to be repaid back to the protocol. It also allows repaying more than actually owed and surplus is tracked as a negative debt for the user's account. However, that surplus is also automatically added to the total lent supply, that can be borrowed, and to the user's unlent balance.

User can then call **lend** function to lend that surplus. However, at that point, it will be added to the **total_lendable_supply** again, inflating its actual value and breaking the protocol's accounting.

core:internal_repay

```
fn internal_repay(&mut self, account_id: &AccountId, asset_amount: Decimal, token_id:
TokenId) -> Result<(), SatoshiError> {
    let account: &mut Account = ensure_some(self.accounts.get_mut(&account_id.clone()),
"Not found");
    let new_balance = account.state.borrowed_balance.withdraw(token_id.clone(),
asset_amount)?;

    // ... other logic ...

    self.total_borrowed_balances -= asset_amount;
    self.total_lendable_supply += asset_amount;  // Surplus added here, can be
double-counted
    // ...
}
```

Recommendation

We recommend either returning the tokens that exceeded the necessary repayment, or adding the surplus to user's lent balance.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-10 Owner Can Arbitrarily Set Asset Price - Medium

The Sailor Lend contract implements an **update_exchange_rate** function which can be used to arbitrarily set the price for any chosen asset. This function can only be called by contract's owner. However, it bypasses the intended flows that leverages Pyth price Oracle. This severely decreases the overall protocol's decentralization level while also increasing the insider threat.

A malicious owner could set a price to arbitrarily chosen value to cause damage to the protocol, specific user, or extract value for own benefit. The **update_exchange_rate** function does not assert the attached deposit of 1 yoctoNEAR, which makes it possible for a compromised owner to execute this function without explicit confirmation.

core:update_exchange_rate

```
pub fn update_exchange_rate(&mut self, base_token_id: TokenId, quote_token_id:
TokenId, price: String, publish_time: I64) {
  self.require_only_owner();
  let price_id = &PriceId { base: base_token_id, quote: quote_token_id };
  let price = Decimal::new(ensure_ok(i64::from_str(&price), "Price attestation invalid
format"), 8);
  let exchange_rate = ExchangeRate { price, publish_time: publish_time.0 };
  self.internal_update_exchange_rate(price_id, exchange_rate);
}
```

Recommendation

We recommend removing this functionality altogether and rely solely on the Price Oracle.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-11 Withdraw Callback Does Not See Errors - Medium

The **withdraw** functionality is withdrawing the tokens from internal accounting and transferring them into **intents** NEAR smart contract using the **ft_transfer_call** flow. It also attaches a **handle_withdraw_callback** callback that checks the promise result and acts accordingly.

However, the result of a previous promise will not be a failure even if the **ft_on_transfer** executed on the **intents** contract will fail.

withdraw.rs

```
pub fn handle_withdraw_callback(&mut self, account_id: &AccountId, token_id: &TokenId,
amount: Decimal) {
    match env::promise_result(0) {
        PromiseResult::Failed => {
            self.handle_withdraw_callback_failure(account_id, token_id, amount);
        }
        PromiseResult::Successful(_value) => {
            // This branch executes even if ft_on_transfer failed!
            self.handle_withdraw_callback_success(account_id, token_id, amount);
        }
    }
}
```

Recommendation

We recommend implementing a withdrawal mechanism using **ft_transfer** flow.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-12 Invalid Approach to Fees and Interest - Medium

Sailor Lend contract implements a mechanism to calculate fees and interest that would need to be paid by borrower and earned by lender. Functions responsible for doing that are **perform_borrower_calculations** and **perform_lender_calculations**. However, they are never called within any of the contract's flow, instead, they should be called individually.

As such, borrower's fees are not actually calculated until someone calls that function for the borrower. Analogously, a specific lender's interest is also not calculated, until someone calls a function to do it.

Consequently, part of the protocol's accounting relies on users calling those functions, especially the debt might not be tracked properly, unless there would be an off-chain bot calling those functions on behalf of users.

As an edge case, it's entirely possible for one user to be both lender and borrower, while also getting benefits of lending without incurring borrowing fees.

Recommendation

We recommend a two step implementation change:

1. Whenever lender or borrower calculations are executed for a given user, make sure that the other form of calculation is done as well. This can be done through wrapping **perform_borrower_calculations** and **perform_lender_calculations** into one function and calling that one instead.
2. Make sure that this single function encompassing both lender and borrower calculations is called each time a specific user tries to perform an action in the protocol.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit **3446235a0abc82377a266bc8c8a056c3523130fc**

GUV-13 No Access Control in Notification Modification - Medium

The Sailor Lend contract implements a liquidation notification system. Each time a user is liquidated, there is an **AccountLiquidationRecord** created and saved in the contract's storage. The record contains a field called **user_notified** and is set to **true** in the **mark_liquidation_notifications_seen** function.

The **mark_liquidation_notification_seen** function lacks access control. It is callable by anyone with an arbitrarily chosen **account_id**. As a consequence, anyone can mark any user's notification (if present) as seen. The notification system is not critical for the protocol's operation, however this lack of access control still can lead to inconsistencies and possible user confusion.

notifications.rs

```
pub fn mark_liquidation_notifications_seen(&mut self, account_id: AccountId) {
    if let Some(records) = self.account_liquidations.get_mut(&account_id) {
        for record in records.iter_mut() {
            record.user_notified = true;
        }
    }
}
```

Recommendation

We advise to review if the notification system is necessary and if so, we recommend implementing an access control mechanism for the **mark_liquidation_notification_seen** function.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-14 update_accounts_at_risk Might Fail to Execute - Medium

The **update_accounts_at_risk** function is meant to loop over all of the accounts saved within the Sailor Lend contract and check their current debt. If they will be over liquidation threshold, those accounts will be added to the **accounts_at_risk** collection.

There are two problems that will prevent this function from executing correctly, at some point:

1. The overall gas limit. NEAR's blockchain configuration currently sets the max available gas that can be used for execution to be 300 TGas units. This is a hard cap and exceeding it will automatically result in OutOfGas error. Because **update_accounts_at_risk** loops over all of the accounts saved in the contract, it will naturally reach that limit at some point, just due to more users using the contract.
2. For each of the loop iteration, there are some logs that are emitted. In NEAR blockchain, there is a hard limit of exactly 100 logs that can be emitted in a single execution. An execution that attempts to emit more automatically results in an error.

Recommendation

We recommend reviewing the protocol's design and considering removing this functionality altogether, as it is not necessary for protocol's operation. Otherwise, we recommend implementing a batching mechanism that will limit the Gas costs and number of logs emitted.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-15 Dead or Useless Code - Informational

In many places throughout the codebase, there are pieces of code that are either unused or don't serve any meaningful purpose. Those are **const** values that are not used anywhere, or functions which were likely used during development process, but have no business purpose. Furthermore, **log** macro is used extensively for **print**-like debugging purposes, which increases the execution cost and time.

Overall, the code residual from the development phase decreases the level of professionalism, and in turn trust, while also possibly increasing the resulting compiled WASM size.

core

```
// Unused constants
pub const SOL: &str = "sol.omft.near";
pub const BTC: &str = "btc.omft.near";
pub const ZEC: &str = "zec.omft.near";
```

core

```
// Development helper functions with no business purpose
#[private]
pub fn helper_call_succesful_promise(&self) -> Promise { ... }
#[private]
pub fn helper_succesful_promise(&self) { log!("Simulating successful promise"); }
#[private]
pub fn helper_fail_promise(&self) { env::panic_str("failure"); }
```

Recommendation

It is recommended to carefully go through the whole codebase and remove unnecessary code.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-16 Invalid ft_transfer Parameters in Borrow - Informational

The **borrow** function, after every check passes successfully, schedules a Promise with **ft_transfer** call to transfer the borrowed stablecoins. It creates the **transfer_args** used as arguments to the fungible token's **ft_transfer** function. Those arguments are incorrect, as they use the **msg** field, which is not one of the parameters defined in **ft_transfer** function's signature. The **msg** is only present in **ft_transfer_call**.

core:borrow

```
let transfer_args = json!({
    "receiver_id": account_id,
    "amount": u128_amount_str,
    "msg": format!("{{"operation": "Borrow","receiver_id":"{}"}}", account_id)
}).to_string().into_bytes();
```

Recommendation

We recommend changing the **msg** parameter into **memo**.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc

GUV-17 Logging Inconsistencies - Informational

We have observed that the logging approach is inconsistent throughout the codebase. Namely, some parts of the code use a struct-based JSON approach to logging, while others simply use strings in **log** macro. It also must be pointed out that some of the crucial functions do not emit any logs, while they should.

Recommendation

We recommend removing unnecessary logs while also adding them to all of the important functions. We also recommend picking one logging scheme and sticking to it for consistency.

Remediation - Fixed

Sailor Lend team has fixed this issue in commit
3446235a0abc82377a266bc8c8a056c3523130fc