Guvenkaya® The Bedrock of Security

Jump Defi

NEAR Rust Smart Contract Security Assessment

Lead Security Engineer: Timur Guvenkaya
Date of Engagement: 11th December 2024 - 13th December 2024
Visit: www.guvenkaya.co

Contents

Findings Details 10

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 Jump Defi

Jump Defi is the only one-stop decentralized finance platform on NEAR Protocol. Jump DeFi lowers the barrier of entry to decentralized finance for users and developers.

Audit Results

Guvenkaya conducted a security assessment of the **Jump Defi smart router** and **Jump farm smart contracts changes** from 11th December 2024 to 13th December 2024. During this engagement, a total of **18 findings** were reported. 4 of the findings were critical, 2 high, 5 medium, and the remaining were either low or informational severity. All major issues were fixed by the Jump Defi.

Project Scope

File name	Link
Lib (Smart Router)	https://github.com/BlockApex/jump-DEFI-contracts/blob/d4d872773163ce82fbd3a0b7f3f2158f294b5cb3/jump-smart-router/src/lib.rs
Constants	https://github.com/BlockApex/jump-DEFI-contracts/blob/d4d872773163ce82fbd3a0b7f3f2158f294b5cb3/jump-smart-router/src/constants.rs
Actions on Farm (Diff)	https://github.com/BlockApex/jump-DEFI-contracts/pull/11/files
Actions on Seed (Diff)	https://github.com/BlockApex/jump-DEFI-contracts/pull/11/files
Simple Farm (Diff)	https://github.com/BlockApex/jump-DEFI-contracts/pull/11/files
Token Receiver (Diff)	https://github.com/BlockApex/jump-DEFI-contracts/pull/11/files

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
11th December 2024	16th December 2024	17th January 2025

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: Token Draining Through Invalid Callback Handling	Severe	Likely	CRITICAL	Moderate	Fixed
GUV-2: Token Draining Through Race Condition	Severe	Likely	CRITICAL	Moderate	Fixed
GUV-3: Token Draining Through Race Condition With Invalid Withdraw Token	Severe	Likely	CRITICAL	Moderate	Fixed
GUV-4: Denial of Service Due to Storage Bloating Via Unlimited Farms	Severe	Likely	CRITICAL	Simple	Fixed
GUV-5: Potential Denial Of Service Due To Storage Bloating	Severe	Possible	HIGH	Simple	Fixed
GUV-6: Unasserted Access Control On Extending Whitelist Postfixes	Moderate	Likely	HIGH	Simple	Fixed
GUV-7: Potential Phishing Opportunity Through Signer Account ID Usage	Moderate	Possible	MEDIUM	Simple	Fixed
GUV-8: Usage of JSON Incompatible Types	Moderate	Possible	MEDIUM	Simple	Fixed
GUV-9: Missing Tests	Moderate	Possible	MEDIUM	Complex	Acknowledged
GUV-10: Missing Prepaid Gas Checking	Moderate	Possible	MEDIUM	Simple	Fixed
GUV-11: Missing Callback On Withdraw	Moderate	Possible	MEDIUM	Simple	Fixed
GUV-12: Usage of Native Collections Instead Of NEAR SDK Equivalents	Moderate	Rare	Low	Simple	Fixed

Finding	Impact	Likelihood	Severity	Remediation Complexity	Remediation Status
GUV-13: Test-Only Function In The Production Binary	Moderate	Rare	Low	Simple	Fixed
GUV-14: Redundant State Check In Initializers	Negligible	Rare	Informational	Simple	Fixed
GUV-15: Redundant Optional Types	Negligible	Rare	Informational	Simple	Fixed
GUV-16: Size Of The Contract Can Be Decreased	Negligible	Rare	Informational	Simple	Fixed
GUV-17: Redundant Withdrawal On Zero Balance	Negligible	Rare	Informational	Simple	Fixed
GUV-18: Hardcoded Contract Accounts	Negligible	Rare	Informational	Simple	Fixed

Findings Details

GUV-1 Token Draining Through Invalid Callback Handling - Critical

The smart router collects user tokens into the contract before performing a swap or allowing withdrawals. After a swap is performed, **check_and_finalize** verifies if the user has sufficient balance through:

smart_router:check_and_finalize:jump-smart-router/src/lib.rs

```
let input_amount: u128 = amount.parse().unwrap();
            let user_balances = self
               .deposits
               .entry(AccountId::from_str(&account_id).unwrap())
               .or_insert_with(HashMap::new);
            let input_balance = *user_balances.entry(input_token_id.clone()).or_insert(0);

          if input_balance < input_amount {
            env::panic_str("Insufficient input balance for the transaction.");
          }
```

A critical vulnerability exists where even if a user lacks sufficient input amount, but the contract has accumulated tokens from other users (since the swap uses the current contract account ID), the panic at this point won't revert the DEX-side swap. This allows a malicious actor to swap out any token amount currently held in the contract.

This vulnerability enables:

- Griefing (other users will lose tokens)
- Manipulating token price (swapping large amount of tokens stored in contract to another one)

POC

Test case to run: **contract_token_draining**

- Construct SwapAction with any token amount up to the total amount held in the contract
- Call ft_on_transfer with a minimal amount (1)
- Note that while our deposits don't increase due to env::panic_str("Insufficient input balance for the transaction."), the tokens are still swapped on the DEX side
- Check ft_balance of current contract to confirm the balance is reduced or zero

Recommendation

Verify that the user has enough funds inside of ft_on transfer per sent number of tokens and SwapAction. Additionally, consider following the pattern of:

- Do all state updates before cross-contract calls
- Inside of the callback, depending on promise result, revert the changes on DEX side

Remediation - Fixed

The Jump Defi team has fixed the issue by creating a reentrancy lock, verifying that provided amount matches the swap action amount, and doing the state updates before cross-contract-calls

GUV-2 Token Draining Through Race Condition - Critical

The smart router collects user tokens into the contract before performing a swap or allowing withdrawals. After a swap is performed, **check_and_finalize** verifies if the user has sufficient balance through:

smart_router:check_and_finalize:jump-smart-router/src/lib.rs

```
let input_amount: u128 = amount.parse().unwrap();
        let user_balances = self
            .deposits
            .entry(AccountId::from_str(&account_id).unwrap())
            .or_insert_with(HashMap::new);
        let input_balance = *user_balances.entry(input_token_id.clone()).or_insert(0);

        if input_balance < input_amount {
            env::panic_str("Insufficient input balance for the transaction.");
        }
```

Additionally, before a user calls swap, the contract verifies sufficient funds through:

smart_router:swap:jump-smart-router/src/lib.rs

```
let amount: u128 = input_amount.parse().unwrap();

        if *current_balance < amount {
            env::panic_str("Not enough amount deposited");
        }
```

However, this verification can be bypassed using a batch call. The contract decreases the balance in the callback, which executes after several blocks, allowing the same amount to be swapped multiple times. Even when a user lacks sufficient input amount, if the contract has accumulated tokens from other users (since the swap uses the current contract account ID), the panic won't revert the DEX-side swap.

This vulnerability enables:

- Griefing (other users will lose tokens)
- Manipulating token price (swapping large amount of tokens stored in contract to another one)

POC

Test case to run: **contract_race_condition_token_draining**

- Deposit certain number of tokens through ft_on_transfer
- Construct SwapAction which matches the token amount you deposited
- Construct batch call to the swap function with the same SwapAction
- Note that while our deposits don't increase due to env::panic_str("Insufficient input balance for the transaction."), the tokens are still swapped on the DEX side
- Check ft_balance of current contract to confirm the balance is reduced or zero

Recommendation

Consider following the pattern of:

- Do all state updates before cross-contract calls
- Inside of the callback, depending on promise result, revert the changes on DEX side

Remediation - Fixed

The Jump Defi team has fixed the issue by creating a reentrancy lock, and doing the state updates before cross-contract-calls.

GUV-3 Token Draining Through Race Condition With Invalid Withdraw Token - Critical

The smart router allows specifying a **withdraw_token** inside the swap function. This token is used to check contract balance before and after the swap to verify fund receipt. It then increases the user's balance per token out in the **check_and_finalize** method:

smart_router:check_and_finalize:jump-smart-router/src/lib.rs

```
let final_balance: u128 = match final_balance_result {
        Ok(value) => value.parse().unwrap_or(0), // Default to 0 if parsing fails
        Err(_) => 0,
    };

    let initial_balance: u128 = initial_balance.parse().unwrap();

    if final_balance < initial_balance {
        env::panic_str("Final balance is less than initial balance, swap might have failed.");
    }

    let tokens_received: u128 = final_balance - initial_balance;

    if tokens_received == 0 {
        env::panic_str("Swap transaction failed, did not receive any funds after swap.");
    }

    match env::promise_result(0) {
        PromiseResult::Successful(_) => {
          let new_balance =
user_balances.entry(output_token_id.clone()).or_insert(0);
          *new_balance += tokens_received;
           ...
    }
```

A vulnerability exists because the withdraw_token is not required to match the output token. This allows a malicious actor to use a cheap token to mint deposits of a more valuable token.

This vulnerability enables:

- Draining of funds from the contract of any token. For example, a malicious actor can use a cheap token during a swap of another token to WNEAR, receiving a large WNEAR deposit that they can then withdraw.

POC

Test case to run: **contract_race_condition_token_minting**

- Construct SwapAction with any token amount and out token being wrap.near
- Swap with withdraw_token being some other cheap token you hold
- Wait 1kms & transfer cheap token to the router contract
- Note that our wrap.near deposit increase to the number of cheap tokens we sent to the contract
- Now we can call withdraw() to withdraw large number of wrap.near

Recommendation

Verify that token out equals to **withdraw_token**. Also, make sure to increase deposit balance by receiving the returned value from DEX instead of relying on **ft_balance_of**.

Remediation - Fixed

The Jump Defi team has fixed the issue by verifying the withdraw token and relying on the swap response instead of ft_balance_of.

GUV-4 Denial of Service Due to Storage Bloating Via Unlimited Farms - Critical

The farm creation process in **ft_on_transfer lacks storage** fee charges, creating a vulnerability to contract bloating. When creating a farm with a long account ID (64 characters) and a seed ID of u16::MAX - 63953_u16 (staying under log limits), each call consumes approximately 13,206 bytes of storage, costing 0.0129N. A malicious actor could execute 100,000 iterations, forcing the contract to spend/lock 1,290N (approximately $9,030) and potentially trigger a denial of service (DoS).

farm:ft_on_transfer:jump-farm/src/token_receiver.rs

```
let terms: HRSimpleFarmTerms = serde_json::from_str(&msg).expect("Invalid msg format
for farm creation");

        // Ensure the reward token matches the token calling ft_on_transfer
        assert_eq!(&env::predecessor_account_id(),terms.reward_token.as_ref(),
          "The reward token must be the same as the token calling ft_on_transfer");

        // Use the provided amount as the initial reward
        let min_deposit = MIN_SEED_DEPOSIT;

        // Create the farm
        let farm_id = self.internal_add_farm(&terms, min_deposit);
```

POC

Test case to run: **storage_bloating_measure**

- Call ft_on_transfer with a long reward account (64 characters) and long seed (u16::MAX - 63953_u16)
- Observe logs to get the number of bytes written to the storage

Recommendation

Implement a storage deposit requirement as in **create_simple_farm**method.

Remediation - Fixed

The Jump Defi team has fixed the issue by asserting the storage inside of the ft_on_transfer.

GUV-5 Potential Denial Of Service Due To Storage Bloating - High

Users are not charged storage fees in **ft_on_transfer**, which could lead to contract bloating. Each ft_on_transfer call with a long account ID (64 characters) adds approximately 125 bytes of storage, costing 0.00122N. A malicious actor could execute 100,000 iterations, forcing the contract to spend/lock 122N (approximately $780) and potentially trigger a denial of service (DoS).

This vulnerability also exists in the swap function, where or_insert is used to create default values for users:

smart_router:swap:ft_on_transfer:jump-smart-router/src/lib.rs

```
let user_balances = self
        .deposits
        .entry(env::signer_account_id())
        .or_insert_with(HashMap::new);
    let current_balance = user_balances.entry(input_token_id.to_string()).or_insert(0);
```

POC

Test case to run: **storage_dos_measure**

- Call ft_on_transfer with a long account (64 characters)
- Measure storage usage and convert it to NEAR

Recommendation

Implement a storage deposit requirement equal to the storage cost of inserting the longest possible account ID.

Remediation - Fixed

The Jump Defi team has fixed the issue by implementing the storage deposit function and asserting the storage inside of the ft_on_transfer.

GUV-6 Unasserted Access Control On Extending Whitelist Postfixes - High

The **extend_auto_whitelisted_postfix** and remove_auto_whitelisted_postfix functions lack proper access control assertions. The **is_owner_or_guardians** method returns a boolean but does not prevent execution when the caller is unauthorized.

amm:extend_auto_whitelisted_postfix:jump-amm/src/owner.rs

```
pub fn extend_auto_whitelisted_postfix(&mut self, postfixes: Vec<String>) {
        assert_one_yocto();
      self.is_owner_or_guardians();
        for postfix in postfixes {
          self.auto_whitelisted_postfix.insert(postfix.clone());
        }
    }
```

amm:remove_auto_whitelisted_postfix:jump-amm/src/owner.rs

```
pub fn remove_auto_whitelisted_postfix(&mut self, postfixes: Vec<String>) {
        assert_one_yocto();
      self.is_owner_or_guardians();
        for postfix in postfixes {
          let exist = self.auto_whitelisted_postfix.remove(&postfix);
          assert!(exist, "{}", ERR105_WHITELISTED_POSTFIX_NOT_IN_LIST);
        }
    }
```

amm:is_owner_or_guardians:jump-amm/src/owner.rs

```
pub(crate) fn is_owner_or_guardians(&self) -> bool {
        env::predecessor_account_id() == self.owner_id
          || self.guardians.contains(&env::predecessor_account_id())
      }
```

These methods control whitelisted_postfixes, which creates critical security risks since an attacker can:

- Add malicious tokens to whitelist
- Remove valid tokens from whitelist

Recommendation

Assert the **self.is_owner_or_guardians()** via:

amm:fix

```
assert!(self.is_owner_or_guardians(), "{}", ERR100_NOT_ALLOWED);
```

Remediation - Fixed

The Jump Defi team has fixed the issue by asserting the check.

GUV-7 Potential Phishing Opportunity Through Signer Account ID Usage - Medium

Throughout the smart contract, **env::signer_account_id** is used both in functions and as arguments for cross-contract calls:

smart_router:swap:jump-smart-router/src/lib.rs

```
#[payable]
    pub fn swap(&mut self, actions: Vec<SwapAction>, withdraw_token: String) -> Promise
{
        let input_token_id =
           AccountId::from_str(&actions[0].swap_action[0].tokens.first().unwrap()).unwrap();

        let input_amount = actions[0].swap_action[0].amounts.first().unwrap();
        let user_balances = self
           .deposits
          .entry(env::signer_account_id())
           .or_insert_with(HashMap::new);
```

smart_router:execute_swap:jump-smart-router/src/lib.rs

```
.then(
            Promise::new(env::current_account_id()).function_call(
               "check_and_finalize".to_string(),
               json!({
                  "initial_balance": initial_balance,
                  "amount": amount,
                  "input_token_id": input_token_id,
                  "output_token_id": output_token_id,
                 "account_id": env::signer_account_id().to_string()
               })
               .to_string()
               .into_bytes(),
               NearToken::from_near(0)
```

While the signer account ID matches the original signer in cross-contract call chains, this creates a potential phishing vulnerability. A malicious actor could trick users into signing a chain of cross-contract calls targeting the smart contract, potentially stealing user funds or accessing privileged functions if the user has admin rights.

Recommendation

Use **env::predecessor_account_id** and pass the original caller within the arguments for callbacks or cross-chain calls instead of relying on **env::signer_account_id**.

Example:

smart_router:execute_swap:jump-smart-router/src/lib.rs

```
#[private]
    pub fn execute_swap(
      &mut self,
      #[callback_result] initial_balance_result: Result<String, near_sdk::PromiseError>,
      msg: String,
     original_caller: AccountId,
     ...
    ) -> Promise {
    ...
      .then(
        Promise::new(env::current_account_id()).function_call(
          "check_and_finalize".to_string(),
          json!({
            "initial_balance": initial_balance,
            "amount": amount,
            "input_token_id": input_token_id,
            "output_token_id": output_token_id,
           "account_id": original_caller.to_string()
          })
          .to_string()
          .into_bytes(),
          NearToken::from_near(0)
```

Remediation - Fixed

The Jump Defi team has fixed the issue by utilizing predecessor account id and passing the caller account inside of the argument.

GUV-8 Usage of JSON Incompatible Types - Medium

The get_deposits function does not use JSON-compatible types for its return value. It uses a plain u128 as the value in the HashMap, when it should instead use U128. Also in InnerSwapAction, the pool_id uses u64 instead of U64.

To preserve interoperability, JSON has a double floating point precision limit. Since double floating-point precision has only 52 bits to represent the mantissa (significant digits during conversion), values must be in the range of [-(2^53)+1, (2^53)-1], with a maximum safe integer of 9007199254740991. Using values larger than 9007199254740991 can cause issues such as:

- 9007199254740991 + 1 == 9007199254740991 + 2
- 9007199254740991**1111 => 900719925474099**00000** (after parsing)
- 77777777777777**7777** => 77777777777777**0000** (after parsing)

These issues can lead to severe logical and calculation bugs.

Recommendation

To handle numbers larger than 2^53-1, input and output data should use string format instead. The NEAR rust SDK provides custom JSON types like U64 and U128: **near_sdk::json_types - Rust**. Alternatively, you can reduce the type to u32 or below, which would work for pool_id.

Remediation - Fixed

The Jump Defi team has fixed the issue by utilizing the correct types.

GUV-9 Missing Tests - Medium

The smart router and farm contracts lack comprehensive test coverage. While some test scripts exist, they don't cover the entire codebase. The farm smart contract, which was forked and modified, is missing its original test suite.

Recommendation

The smart router requires both unit tests and integration tests using near-workspaces. For the farm contract, the original test suite should be forked. When modifying the forked farm contract, verification must include:

- Running the complete forked test suite
- Confirming that any test failures are directly related to the modifications and checking for unintended effects on the system
- Addressing test failures related to the modifications, and if systemic issues are discovered, conducting additional verification and implementing necessary fixes

Remediation - Acknowledged

The Jump Defi team has acknowledged the issue and will fix it later.

GUV-10 Missing Prepaid Gas Checking - Medium

The smart router contract lacks prepaid gas validation before making cross-contract calls. This creates a risk where there may be sufficient gas to execute the initial cross-contract call but not enough for its callback.

Recommendation

Calculate and verify available gas against the worst-case scenario before each cross-contract call by subtracting used gas from prepaid gas.

Remediation - Fixed

The Jump Defi team has fixed the issue by asserting the gas usage.

GUV-11 Missing Callback On Withdraw - Medium

The smart router contract is missing a callback after the **ft_transfer** in the **withdraw** function. The user's balance is decreased before the cross-contract call, which means if the transfer fails, the user cannot withdraw their funds again.

smart_router:withdraw:jump-smart-router/src/lib.rs

```
if *current_balance < amount.0 {
        env::panic_str("Not enough amount deposited for withdraw");
    }

    Promise::new(AccountId::from_str(token_id.as_str()).unwrap()).function_call(
      "ft_transfer".to_string(),
      json!({"receiver_id": env::signer_account_id(), "amount": amount.to_string(),
"msg": ""}).to_string().into_bytes(),
      NearToken::from_yoctonear(1),
      Gas::from_tgas(35),
    );
```

Recommendation

Implement a callback for the **ft_transfer** call to revert the balance decrease if the transfer fails.

Remediation - Fixed

The Jump Defi team has fixed the issue by adding the callback.

GUV-12 Usage of Native Collections Instead Of NEAR SDK Equivalents - Low

The contract uses a nested HashMap rather than utilizing LookupMap or IterableMap from the NEAR SDK store collections.

smart_router:jump-smart-router/src/lib.rs

```
pub struct DepositTracking {
deposits: HashMap<AccountId, HashMap<String, Balance>>,
fee: u32,
admin: AccountId,
}
```

Native collections are not optimized for handling large amounts of data. As a native collection grows, deserializing it from memory becomes increasingly gas-intensive. If the collection becomes too large, the contract may consume all available gas while attempting to read its state, causing all function calls to fail.

Recommendation

Consider utilizing collections from the **near_sdk::store**.

Remediation - Fixed

The Jump Defi team has fixed the issue by utilizing the collections from the near_sdk::store.

GUV-13 Test-Only Function In The Production Binary - Low

The **reset_tokens** function is considered as test-only, but lacks the**#[cfg(test)]** directive. Without this directive, the function will be included in the final WASM binary, unnecessarily increasing the binary size and expanding the attack surface.

smart_router:reset_tokens:jump-smart-router/src/lib.rs

```
    #[private]
    pub fn reset_tokens(&mut self, account_id: String) {
        // Ensure the caller is the admin
        assert_eq!(
            env::signer_account_id(),
            self.admin,
            "Only the admin can reset tokens."
        );

        let account = AccountId::from_str(&account_id).expect("Invalid account ID
format");
        ...
```

Recommendation

Consider either removing the test function or marking it with**#[cfg(test)]**.

Remediation - Fixed

The Jump Defi team has fixed the issue by marking the function with #[cfg(test)] .

GUV-14 Redundant State Check In Initializers - Informational

We noticed that the initializer in the smart contract uses a redundant state check - **assert! (!env::state_exists(),"The contract has already been initialized");**

smart_router:new:jump-smart-router/src/lib.rs

```
#[init]
#[payable]
pub fn new() -> Self {
 assert!(
   !env::state_exists(),
   "The contract has already been initialized"
 );
  ...}
```

Recommendation

Consider removing the state check, as the **#[init]** macro already handles it.

Remediation - Fixed

The Jump Defi team has fixed the issue by removing the redundant state check.

GUV-15 Redundant Optional Types - Informational

The **CombinedAction** struct uses optional (Option) types for its actions and withdraw fields. However, these fields are required and are directly unwrapped throughout the codebase.

smart_router:jump-smart-router/src/lib.rs

```
pub struct CombinedAction {
  withdraw: Option<WithdrawAction>,
  swap_actions: Option<Vec<SwapAction>>,
}
```

Recommendation

Consider removing the optional types to make the code's intent more explicit.

Remediation - Fixed

The Jump Defi team has fixed the issue by removing redundant optional types.

GUV-16 Size Of The Contract Can Be Decreased - Informational

We noticed that the crate type is set as both **cdylib** and **rlib**. However, since NEAR smart contracts are compiled to WASM, **rlib** is unnecessary. Eliminating **rlib** can significantly reduce the size of the generated WASM binary.

smart_router:cargo:jump_smart_router/Cargo.toml

```
[package]
name = "jump_smart_router"
version = "0.1.0"
edition = "2018"

[dependencies]
near-sdk = "5.1.0"
borsh = "0.9.1"
serde = { version = "1.0", features = ["derive"] }

[lib]
crate-type = ["cdylib", "rlib"]

[profile.release]
codegen-units = 1
# s = optimize for binary size ("z" would additionally turn off loop vectorization)
opt-level = "z"
...
```

Recommendation

Consider removing rlib from Cargo.toml.

Remediation - Fixed

The Jump Defi team has fixed the issue by removing the rlib.

GUV-17 Redundant Withdrawal On Zero Balance - Informational

We noticed that in **withdraw_tokens** if current balance is zero contract still does the cross-contract call. It is redundant and wastes resources

smart_router:withdraw_tokens:jump-smart-router/src/lib.rs

```
            if *current_balance > 0 {
                *current_balance -= amount;
            }

            Promise::new(AccountId::from_str(token_id.as_str()).unwrap()).function_call(
                "ft_transfer".to_string(),
                json!({"receiver_id": env::signer_account_id(), "amount": amount.to_string(), "msg":
""}).to_string().into_bytes(),
                NearToken::from_yoctonear(1),
                Gas::from_tgas(35),
            );
```

Recommendation

Consider rewriting the logic to be something like:

smart_router:withdraw_tokens:jump-smart-router/src/lib.rs

```
 #[payable]
pub fn withdraw_tokens(
    &mut self,
    token_id: String,
   amount: Option<U128>,
) -> PromiseOrValue<U128> {
    let user_balances = self
        .deposits
        .entry(env::predecessor_account_id())
        .or_insert_with(HashMap::new);

    let current_balance = user_balances.entry(token_id.to_string()).or_insert(0);

   if *current_balance == 0 {
      env::panic_str("nothing to withdraw");
   }

   let amount = amount.unwrap_or(U128(*current_balance));

   if *current_balance < amount.0 {
      env::panic_str("Not enough amount deposited for withdraw");
   }

    Promise::new(AccountId::from_str(token_id.as_str()).unwrap()).function_call(
        "ft_transfer".to_string(),
        json!({
            "receiver_id": env::predecessor_account_id(),
            "amount": amount.to_string(),
            "msg": ""
        }).to_string().into_bytes(),
        NearToken::from_yoctonear(1),
        Gas::from_tgas(35),
    );...
```

Remediation - Fixed

The Jump Defi team has fixed the issue by rewriting the logic.

GUV-18 Hardcoded Contract Accounts - Informational

The contract uses hardcoded contract accounts for external contract interactions. This creates maintenance issues since any change to these contract addresses would require redeploying the entire contract.

smart_router:constants:jump-smart-router/src/constants.rs

```
pub static REF_EXCHANGE: &str = "v2.ref-finance.near";

pub static REF_EXCHANGE_TEST: &str = "ref-finance-101.testnet";

pub static JUMP_EXCHANGE: &str = "pools.jumpfinance.near";

pub static JUMP_EXCHANGE_TEST: &str = "jump_amm.testnet";

pub static VEAX_EXCHANGE: &str = "veax.near";

pub static VEAX_EXCHANGE_TEST: &str = "veax-dex15.testnet";
```

Recommendation

Contract accounts should be set through the initializer during deployment, with setter functions implemented to allow updates when needed.

Remediation - Fixed

The Jump Defi team has fixed the issue by removing the constants file